CMU-CS-81-113

12

Mechanical Transformation of Task Heuristics into Operational Procedures

Ph.D. Thesis

David J. Mostow
Computer Science Department
Carnegie-Mellon University

14 April 1981

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

(14) CMU-CS-81-113

(6)

Mechanical Transformation of Task Heuristics into Operational Procedures

Ph.D. Thesis

(9) Doctoral thesis

(10) David J. Mostow
Computer Science Department
Carnegie-Mellon University

(11) 14 April 1981

(12) 511

(15) F33615-78-C-1551, ARPA Order-3597

This research was sponsored in part by a National Science Foundation Graduate Fellowship, in part by the National Science Foundation under grant no. MCS77-03273 to the Rand Corporation, and in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551.

403 081

[illegible]

Abstract

Advice about how to perform a task can often be expressed in the language of the task domain quite simply. For example, advice for how to play the card game Hearts includes "avoid taking a trick with points," "try to flush out the Queen of spades," and "don't lead a high card in a suit where an opponent is void." However, such advice is not *operational* because it is not expressed in terms of the basic capabilities of the task agent -- it doesn't tell what card to play next. Developing a procedure to implement the advice ("operationalizing" the advice) can be difficult due to limitations imposed by the structure of the task and the environment in which the task is performed. Advice may recommend an action that cannot be directly executed -- a Hearts player cannot flush the Queen by snatching it from an opponent's hand. Advice may be expressed in terms of unobservable information -- a Hearts player cannot peek at opponents' cards to see who's void, or look into the future to find out if a trick will have points.

This dissertation characterizes the operationalization of advice as a series of problem transformations leading from the advice statement to a procedure for achieving a specified goal (avoid taking points) or evaluating a specified quantity (decide whether an opponent is void). It describes some general operators for performing such transformations. These operators have been implemented in a program called FOO as domain-independent transformation rules that access a knowledge base of task domain concepts. They have been used to operationalize advice for the card game Hearts and a music composition task. Some of FOO's transformation rules represent high-level strategies for operationalizing advice. Additional rules represent reasoning methods used to reformulate advice in terms of these operationalization strategies, solve the subproblems they generate, and translate their results into a usable form. Although FOO lacks a problem-solving component for choosing which of these transformation rules to apply, the dissertation shows how means-end analysis could be used to guide the search through the problem space.

The dissertation formalizes the notion of reformulating one concept in terms of another, for example, reformulating "take points" in terms of "playing the highest card in the suit led." It describes mechanical techniques for mapping domain-specific problems onto general methods. In particular, it examines in detail the process of formulating a problem as a heuristic search, and presents domain-independent rules that improve a search procedure by deriving new heuristics based on analysis of the problem and knowledge about the task domain.

A

Table of Contents

List of Figures

Preface

Papers and talks in artifical intelligence tend to include buzzwords and "motherhood statements" about blackboard organizations, knowledge sources, hierarchical levels of representation, multiple perspectives, and so forth. This phenomenon is by no means limited to AI; it characterizes other areas of computer science as well, and no doubt numerous other fields. Short research presentations all too often contain nothing else, and provide examples of *content-free language.*

The spectacle of an uninformative presentation made by an intelligent researcher poses a genuine paradox. One explanation for this paradox is that the real problem involved in most AI research is to take an existing, more-or-less self-evident (or at least widely accepted as plausible) approach and *operationalize* it for a particular task. For example, one might use the blackboard model to understand speech [Erman 75], decode proteins [Engelmore 79], or learn the rules of baseball [Soloway 77][1]. However, the real content of such an effort lies in figuring out how to make the general paradigm operational in the context of the particular problem:

> What are the sources of knowledge?
>
> How should this knowledge be represented?
>
> What language should the knowledge sources communicate in?
>
> How should their interactions be organized?
>
> What policies should govern the allocation of computational resources, and what mechanisms should be used to implement them?
>
> How should the problem be defined, and how should a potential solution be evaluated?

These are the hard problems of research, the kind that often lie just below the level at which the research is reported because their solutions are difficult to communicate. This train of thought leads to the provocative assertion that

> *Operationalization is the real content of most AI research.*

[1]The use of these projects as examples is not meant to reflect in any way on the presentational skills of their authors.

This assertion can be broadened to include a considerable proportion of scientific research, or even intelligent behavior in general; however, it applies with special force to computer science, since this field is to such a great extent a *study of methods.* It is illustrated by the fact that most of the dissertation research reported here was directed at operationalizing the concept of operationalization. The assertion implies that

> *Real progress in automating the construction of intelligent programs will ultimately require a thorough understanding of operationalization.*

This dissertation is one step toward that goal.

Acknowledgments and Dedication

This dissertation owes a great deal to many people; I can claim exclusive credit only for the errors it contains.

Each member of my thesis committee contributed in a different but very helpful way. As my thesis supervisor, Rick Hayes-Roth suggested the initial problem based on his strong intuitions about the potential role of machine-assistance in heuristic programming, and provided enthusiasm and ideas when, as so often happened, I got bogged down in details and lost view of the broader picture. As my advisor, Rick has served above and beyond the call of duty; I am grateful to him for encouraging me to publish lots of papers (sometimes by writing them himself!), arranging for me to work on my research at Rand, giving me the opportunity to attend stimulating conferences and workshops, bringing attention to my work, and generally guiding my intellectual development as a computer scientist.

Allen Newell's excellent high-level suggestions helped define the form of the project, cut years of work from it, and focus it on the important scientific issues. As chairman of my committee, he sheltered me from the administrative details so I could concentrate on the dissertation. Jaime Carbonell is one of the best operationalizers I know, and gave generously of his time to help me operationalize "operationalization." Bob Balzer's sharp insight led to questions whose answers were hard to find but invariably increased my understanding of the issues.

I am grateful to Carnegie-Mellon, Rand, and Stanford for the research support they have provided, in money, space, computational resources, and socially and intellectually stimulating environments. In particular, my understanding of operationalization was clarified by discussions with Jim Bennett, Mike Genesereth, Phil Klahr, Don Kosy, Doug Lenat, Stan Rosenschein, and others. I owe special thanks to Stan for enlightening me on the importance of understanding the semantics of one's knowledge representation, and to Jim for improving the presentation of ideas.

The dissertation was composed using the Bravo text editor and Alto personal computers donated by the Xerox Corporation. Bruce Nelson's BRIBE program translated my Bravo files into input for the SCRIBE document production system, developed by Brian Reid and now supported by Unilogic. I want to thank Bruce, Brian, and Unilogic both for the usefulness of their programs and the generosity of their assistance in using them.

I owe my career in computer science to a series of circumstances. First is the combination of curiosity and determination my parents have always tried to instill in me. Their loving influence has followed me through my adventures since leaving home. I often resisted their advice, but it just as often turned out to be helpful, and I am grateful for it. My brother Mark gave me my first few programming lessons, and more generally has imparted to me many useful heuristics that he, as firstborn, had to discover the hard way. Sid Trachtenberg, head of the math department at Amity Regional High School, Woodbridge, Connecticut, encouraged my newfound interest in computing and gave me the opportunity to pursue it: he managed to borrow a PDP-8 computer for the school, and then talk a conservative school board into buying it for $27,000 in 1968 dollars, based on his faith that the kids who played with it would go on to develop their interest in useful directions. The three years I spent playing with that machine -- with its vintage teletype for I/O, 4K of 12 bit words for primary memory, and paper tape for secondary storage -- certainly got me hooked, and here I am.

My N years at Carnegie-Mellon have been greatly enriched by some wonderful people who have given me a home away from home. It is largely because of the Eisners (and Grandma Fuffy!) that I will miss Pittsburgh.

Last and most: I am indebted to a very special person who has given me the courage to face and finish this project, to grapple with a series of depressingly ill-defined problems, and to keep growing into a more human being. With gratitude and love I dedicate this dissertation:

To Marjie, of course!

Chapter 1 Introduction

A central theme of recent research in artificial intelligence is that

Intelligent task performance requires large amounts of knowledge about the task domain.

A major bottleneck in building a program to perform an intelligent task lies in converting knowledge into a form readily applicable to the task by a relatively simple runtime mechanism, such as an EVAL routine for LISP procedures [McCarthy 63], an interpreter for condition-action productions [Davis 76], or a back-chaining inference engine for MYCIN-style rules [Davis 77a].

Machine-aided heuristic programming [Hayes-Roth 78a] is a proposed approach for facilitating the translation of domain knowledge into task performance. The idea is to instruct a computer to do a new task in much the same way one would instruct a person: by giving an informal description of the task plus some advice for how to do it well. For example, if the task is to play the card game Hearts (described in Appendix A.1), the task description would include the rules of the game, and the advice would include heuristics like:

"Avoid taking points."

"Don't lead a high card in a suit in which an opponent is void."

"If an opponent has the Queen of spades, try to flush it."

If the task is to compose a *cantus firmus* (*q.v.* Appendix A.2), the basic description would be "generate a series of musical tones of equal length," and the advice would include heuristics like these (taken from a textbook on counterpoint):

"The cantus should include an unrepeated climax."

"The *melodic range* of the cantus should not exceed a major tenth."

Since these descriptions are expressed in terms specific to the respective task domains of Hearts and music, rather than in the vocabulary of programming languages, it would also be necessary to provide the computer with definitions of these terms, *e.g.*:

"Being *void* in a suit means having no cards in that suit."

"*Flushing* a card means forcing whoever has it to play it."

"The *melodic range* of a tone sequence is the interval between its lowest and highest notes."

Converting this kind of task description into effective behavior involves several processes. The informal natural language description must be *encoded* into a precise representation of its meaning better suited to mechanical manipulation. It must be *operationalized* in terms of primitive capabilities. Different pieces of advice must be *integrated*. The knowledge must be *applied* to specific task situations at runtime.

This paradigm avoids re-operationalizing the task description for each new task situation; rather, it converts the task description into an operational form that is general enough to be applied to different task situations. One reason for this is *efficiency*. If operationalization is computationally expensive, performing it repeatedly for a frequently performed task is intolerable. Another reason is *timeliness*. Applying advice to a task situation can require some advance preparation. For example, consider the advice

"Don't lead a high card in a suit in which an opponent has shown void."

This advice bears on the decision of what card to lead, but it requires remembering who has shown void in what suit. A human can try to rely on serendipity or on total recall for remembering such information, but a more practical approach for a computer program with limited memory is to operationalize the advice ahead of time and decide to remember who shows void in what suit.

1.1. Problem Definition

This dissertation focusses on the problem of transforming advice into an *operational* form, *i.e.*, a procedure that can be applied to the task at hand:

> A procedure is a (relatively) operational form of a piece of advice if the agent performing the task can apply the procedure in (many of) the situations where the advice is relevant, and the result of doing so is (often) consistent with the advice.

Thus operationality is a relation between a piece of advice, the agent for whom the advice is intended, and a procedure executed by the agent to implement the advice. Whether such a procedure is operational may depend on the *actions* the agent can perform in the task environment, the *information* the agent can obtain by direct observation, and the *computations* the agent can make based on that information.

1.1.1. Model of a Task Agent

This view is based on a model in which an agent's interactions with a task environment are restricted to a channel defined by the agent's capabilities for observation and action, as shown in Figure 1-1.

```
           <--- store ---              --- function arguments -->
MEMORY                      TASK AGENT                            COMPUTATION
           --- recall -->              <---- function value -----
                           ↑         |
                           |         |
                       sensory  effector
                    operations  operations
                           |         |
                           |         v

                       TASK ENVIRONMENT
```

Figure 1-1: Model of a task agent

Sensory operations obtain information by direct observation of the task environment. The agent's sensory capabilities can be modelled by a collection of functions $f_1, ..., f_n$ whose domain is the space of task situations, or *states*. The agent knows $f_1(s), ..., f_n(s)$ in any state s, and this information covers everything it can directly observe about s. Each such function will be called *observable*. For example, in Hearts one can observe one's hand and the cards in the pot. Note that the agent has no direct access to the "value" of the situational variable s, which denotes the entire state of the environment at a given time. In FOO, the operationalization program to be described later, this variable is not referred to explicitly; instead, state-dependent expressions like (CARDS-IN-POT) ("the cards in the pot") and (HAS ME C1) ("player ME has card C1") are *implicit* functions of s.

The other direction along the channel between agent and task environment consists of *effector* operations -- the agent's capabilities for performing actions that change the state of the task environment. The actions in Hearts include playing a card (from one's hand to the pot) and taking a card (from the pot to one's pile). Such an operation can be modelled by a function f from states to states, where f(s) is the state that results from performing the operation f in state s. Such a function will be called *doable* if it corresponds to an action the agent can perform. The function is only defined on those states in which the agent can perform the action. Similar functions may represent actions that other agents in the environment can perform, but these are not (directly) doable. Thus the action (PLAY ME QS) ("player ME plays the Queen of spades") is doable whenever player ME (the agent) can legally play the Queen of spades, but the action (PLAY P1 QS) is not doable for P1 ≠ ME, since the actions of player P1 are not under player ME's direct control.

In addition to sensory and effector operations, which comprise the two directions of the channel between agent and environment, the agent has purely internal capabilities. One of these is *computation*: the agent can calculate the values of various functions given their arguments. Such functions will be called *computable*. Another capability is (unlimited) *memory*: in principle, the agent has access to any past observation or computation. (This capability can be implemented without prohibitive storage costs if it is known *a priori* what information will be needed later (§2.4).)

The class of evaluable state-dependent functions in this model can be recursively defined:

1. *Direct observation:* Any observable function f is evaluable.
2. *Computation:* If f is evaluable and g is computable, then $g \circ f$ is evaluable.
3. *Memory:* If f is evaluable, then the value f_t that f had at a past time t is evaluable.

This paradigm incorporates several simplifications:

1. The model does not specify computational costs, although such costs can certainly affect the feasibility of evaluating a given expression.
2. Memory processes for storage and retrieval are treated implicitly rather than explicitly; there is no concept of a data structure and operations on it.
3. The agent's "state of mind" is not part of the task environment; otherwise, the distinction between sensory and state-changing operations would break down, since the act of making an observation affects the agent's state of knowledge.

In particular, the paradigm does not allow for an explicit model of the environment maintained by the task agent by applying domain-specific inference rules to data (including both direct observations and previous inferences) and updating the model to incorporate the results. In this alternative paradigm, advice that refers to the environment is evaluated relative to the model. For instance, one inference rule for Hearts is "If suit S is led and player P plays a card in some other suit, assert that player P is void in suit S." The task agent would apply this rule upon observing an opponent break suit. To follow the advice "Don't lead a high card in a suit where an opponent is void," the task agent would search its model for assertions of the form "player P is void in suit S" and choose a suit for which it found no such assertions.

This is actually an approach to the *integration* problem mentioned earlier: it assumes that translating domain knowledge into forward inference rules, using them to maintain a model of the environment, and evaluating advice relative to the model will lead to task performance consistent with the advice. In contrast, the approach taken in this dissertation deduces information as needed (although some of the operationalization strategies involve explicit decisions to remember specific kinds of information for later use). The two approaches are closely related: many of the techniques developed in the dissertation could be used to translate domain knowledge into forward inference rules.

In the model of Figure 1-1, a piece of advice can be non-operational for the task agent for any of several reasons. It may require evaluating a quantity defined in terms of *unobservable* data or an *unpredictable* future event. Similarly, it may require achieving a goal that is not expressed as a *doable* action or that depends on an *uncontrollable* choice made by some other agent.

1.1.2. Definition of Operationalization

Suppose the agent is a computer program with a few rudimentary card-playing capabilities: it has procedures for observing the cards in its hand, deciding whether it can legally play a card, and moving a card. It has no inferential ability, but it has a mechanism similar to LISP's EVAL for executing procedures and evaluating expressions, augmented by a few features like three-valued logic (true, false, and unknown) and the ability in a given task situation to try alternative evaluation methods (partial definitions) for a given function to find one that works (returns a result). Assume also that it can deal with historical references, *i.e.*, can remember the value that an expression had at a specified time in the past, how many times a described event occurred over a specified time span, and so forth.

What does it mean to operationalize Hearts advice for such an agent? Typically, it involves figuring out how to *achieve* some condition or *evaluate* some quantity. For example, to make the advice "avoid taking points" operational for the assumed agent, it must be converted into a plan like "play a low card." This is a plan the agent can execute, assuming it has a test for deciding whether a card is low. Thus the basic problem in operationalizing "avoid taking points" is of the form:

Find an executable plan to *achieve* a desired condition.

Another kind of operationalization problem is:

Find an effective method to *evaluate* a relevant quantity.[2]

[2]One could rephrase such a problem as "Achieve a state in which I know the value of the quantity," but this approach runs into the difficulties associated with explicitly representing the idea of knowing something (§4.1) (§8.3.2).

For example, consider the advice "don't lead a high card in a suit in which an opponent is void." The agent is not allowed to look at its opponents' cards. Thus the operationalization problem here is to find a procedure for guessing if an opponent is void in a given suit.

Both kinds of problems may arise in operationalizing a single piece of advice, such as:

"If an opponent has the Queen of spades, try to flush it."

Following this advice requires *evaluating* whether an opponent has the Queen of spades (without looking at opponents' cards!) and *achieving* the goal of forcing the opponent to play it.

The operationality of advice is affected by the *information available* to the agent receiving it. For example, the condition "an opponent has the Queen of spades" is not decidable by direct observation in Hearts since one is not allowed to look at one's opponents' cards. Operationalizing this condition means devising a way to evaluate it. The condition "the cantus must contain an unrepeated climax" is non-operational for a left-to-right sequence generator, since it depends on the complete sequence, not just the portion that's been generated so far.

Operationality can depend on the *behavioral limitations* of the task agent. For example, the advice "flush out the Queen of spades" can't be satisfied by plucking the Queen from the hand of the opponent holding it, since the rules of Hearts restrict players to playing cards from their own hands.

Operationality can depend on the *computational limitations* of the task agent. For example, "predict the consequences of playing a card by enumerating all possible card sequences for the round" is a computationally infeasible way to avoid taking points. Similarly, "generate all tone sequences of the desired length and pick one that satisfies the other constraints" is not a feasible way to generate a cantus firmus. Thus operationalization includes the problem of making *theoretically* executable computations efficient enough to be *feasible*. The distinction between this and what an optimizing compiler does is partly one of degree. The term "operationalization" applies to quantum speed-ups based on high-level understanding of the task, rather than to small improvements based on local transformations of code. Moreover, such speed-ups may be achieved by settling for an approximation of the original advice -- such as a procedure that is only sometimes executable, or occasionally produces an incorrect result. This approach is not permissible for a faithful compiler.

Operationality can be *relative*. A procedure is a more or less operational form for a piece of advice according to

1. Its *applicability:* how often the agent can execute the procedure.
2. Its *effectiveness:* how often executing the procedure produces the desired result.

For example, consider two operationalizations of the advice "avoid taking a trick with points":

1. "Play your lowest legal card."
2. "Underplay another card in the trick."

The first is always *applicable* but not always *effective*, since the agent's lowest card may turn out to be the highest card played in the trick. In contrast, the second is guaranteed to be *effective*, but is only *applicable* when the agent has a card lower than a card already played in the trick.

To restrict the problem, this dissertation makes certain assumptions. Since natural language interpretation is beyond the scope of this work, the initial task description and advice are assumed to be encoded in an *unambiguous internal representation.* To avoid the problems associated with integrating different pieces of advice, the dissertation only considers pieces of advice that can be operationalized *independently* of each other, and then integrated subsequently, say as different terms in an evaluation function. Thus each piece of advice is encoded as an *expression* that may refer (implicitly or explicitly) to various aspects of the task, but not to other pieces of advice. This restriction excludes advice whose operationalization depends on the other advice given, *e.g.*:

> "Play a card that satisfies more than one goal."
> "Assume an opponent's action is motivated by the same reason you'd have for that action."

Explicit methods for achieving or evaluating a condition are also excluded, since such advice is already operational, *e.g.*:

> "A suit is safe for the first two tricks led in it."
> "To avoid taking points, play a low card."

These considerations lead to a definition of *(partial) operationalization* as:

> *Transformation of a well-defined expression into a procedure that will (often) be feasible for a specified mechanism to execute using the data and actions available to it at execution time, and whose execution will (often) produce the result described by the original expression.*

1.1.3. Thesis of the Dissertation

The goal of this dissertation is to explore the kinds of knowledge useful in operationalizing advice. The underlying thesis is that

> *There exist general operationalization methods that can be stated independently of any specific task domain, although applying them generally requires domain-specific knowledge.*

1.2. Method of Investigation

This thesis was tested by following the plan of research described below.

1. Select a task domain. (§1.2.1)
2. Construct a problem space for operationalization. (§1.2.2)
3. Generate example problems, operators, and solution paths in this space. (§1.2.3)
4. Test generality on examples from a second task domain. (§1.2.4)
5. Test operationality of generated solutions. (§1.2.5)

1.2.1. Selecting a Task Domain

The task of playing the card game Hearts was selected as a good vehicle for exploring operationalization. The Hearts environment is easily to model, and the rules of legal play are simple, so the overhead of using it as an experimental vehicle is small compared to tasks requiring large amounts of specific world knowledge, such as understanding international relations [Carbonell 78] or planning synthesis experiments in molecular genetics [Martin 77].

Although the game of Hearts is easy to describe, it is hard to win. Hearts involves multiple agents, each of whom has partial information (you can't see your opponents' hands) and partial autonomy (you can play a card only from your own hand, only in your own turn, and only if it's a legal card for you to play). Thus it contrasts with, say, the blocks-world task, where there is a single agent with a complete model of the world and the freedom to perform any physically possible operation at any time.

There is no apparent algorithm for winning at Hearts, but a varied collection of advice on how to play well provides a rich source of operationalization problems. For example, operationalizing the advice "don't lead a suit in which an opponent is void" involves devising a way to decide if an opponent is void in a given suit.

Appendix A.1 describes the rules of Hearts and the advice use in later examples.

1.2.2. Building a Problem Space

A problem space for operationalization was built, consisting of a representation and a set of operators on it. The states in the problem space are expressions in an unambiguous LISP-like language, described in (§1.3). The operators are rules that transform expressions. Given a piece of advice encoded as an expression, the problem is to find a sequence of rules that transforms it into an operational expression, where "operational" is defined relative to the assumed runtime interpreter. A sequence of expressions leading via such transformations from an initial piece of advice to an operationalization of it is called a *derivation.*

The 300 or so transformation rules are implemented in a 300K LISP program called FOO, for "First[3] Operational Operationalizer." The rules are the major product of the research, *i.e.*, most of the dissertation consists of statements about them. They are *general* in that they don't refer directly to the task domain, but they access a domain knowledge base. The problem space is discussed in more detail in (§1.2.7), and the representation of domain knowledge is described in (§1.3).

FOO has no problem-solving component: it can apply a rule, but it doesn't know what rule to apply to a given expression, or even when to halt because the expression is operational. Thus the dissertation factors operationalization into two problems -- formulating the problem space and solving problems in this space -- and neglects the second in order to concentrate on the first. It demonstrates that mechanical reasoning paths exist in the space, but does not solve the search control problem involved in finding such paths. (§7) proposes an approach to this problem.

1.2.3. Generating Examples

I encoded several pieces of advice as expressions and operationalized each one by applying a sequence of transformation rules, adding new rules as needed. At each point in the sequence, I chose which rule to apply and which sub-expression to apply it to, but the actual application was performed by FOO. In cases where applying a rule involved making some selection, *e.g.*, choosing from a set of concepts retrieved by FOO from its knowledge base, I made the selection. I also decided when to declare an expression operational and terminate the sequence.

[3]Or "Fairly," or "Faintly."

The derivations of operational forms for Hearts advice generated in this interactive mode are designated DERIV2 through DERIV12, and appear in Appendix D. (§1.2.3.1) summarizes the derivations by showing the initial and final expressions and mentioning the kernel ideas. For example, in DERIV3, "flush the Queen" is operationalized as "keep leading spades until the Queen is played" by applying a general plan for depleting a set. (§1.2.3.2) describes the criteria used to select the advice operationalized in the derivations.

1.2.3.1 Derivation Summaries

The derivations generated using FOO are designated DERIV2 through DERIV14, and are described below.

DERIV2 operationalizes "avoid taking points" as a heuristic search procedure (§3) that tries to find a scenario in which playing a given card leads to taking points.

```
DERIV2 #STEPS 105 --DOMAIN: HEARTS
        --PROBLEM: (AVOID-TAKING-POINTS (TRICK))
        --SOLUTION: heuristic search in space of card sequences
```

DERIV3 operationalizes "flush the Queen of spades" as "keep leading spades" by instantiating a general plan for depleting a set (§2.5), analyzing the rules of legal play, and applying knowledge about constraining someone else's choices (§2.7).

```
DERIV3 #STEPS 93 --DOMAIN: HEARTS
        --PROBLEM: (ACHIEVE (FLUSH QS))
        --SOLUTION: (UNTIL (PLAYED! QS) (ACHIEVE (LEAD-SPADE ME)))
```

DERIV4 uses the pigeon-hole principle (§2.2) and case analysis (§5.7.1) to operationalize "the Queen of spades is out" as "the Queen is not in the pot, it's not the hole card, I don't have it, and it hasn't been taken."

```
DERIV4 #STEPS 55 --DOMAIN: HEARTS
       --PROBLEM: (EVAL (QS-OUT))
       --SOLUTION:
        (EVAL (NOT (OR [IN-POT QS] [AT QS HOLE] [HAS-ME QS] [TAKEN QS])))
```

DERIV5 modifies the plan derived in DERIV3 for flushing the Queen so as to avoid taking the Queen oneself. This example illustrates a strategy for integrating multiple goals (§2.10).

```
DERIV5 #STEPS 53 --DOMAIN: HEARTS
       --PROBLEM: (AND [ACHIEVE (FLUSH QS)] [AVOID (TAKE ME QS) (TRICK)])
       --SOLUTION: (UNTIL (PLAYED! QS) (ACHIEVE (LEAD-SAFE-SPADE ME)))
```

DERIV6 operationalizes the advice "avoid taking points" as "play a low card" by reformulating it in terms of a choice variable (§4). A variant solution, "underplay the King of diamonds," assumes a situation in which the King of diamonds has been led. This example illustrates that operationalizing advice in the context of a specific task situation may produce a more effective but less general solution.

```
DERIV6 #STEPS 62 --DOMAIN: HEARTS
      --PROBLEM: (AVOID-TAKING-POINTS (TRICK))
      --SOLUTION:
        (ACHIEVE (=> (AND [IN-SUIT-LED (CARD-OF ME)] [TRICK-HAS-POINTS])
                     (LOW (CARD-OF ME))))
      --VARIANT-SOLUTION:
        (ACHIEVE (=> (AND [IN-SUIT-LED (CARD-OF ME)] [TRICK-HAS-POINTS])
                     (LOWER (CARD-OF ME) DK)))
```

DERIV7 operationalizes "get the lead" as "play a high card in the suit led." The reformulation of "win the trick" as a predicate on player ME's card is borrowed from DERIV6.

```
DERIV7 #STEPS 8 --DOMAIN: HEARTS --PROBLEM:
        (ACHIEVE (LEADING ME))
      --SOLUTION:
        (ACHIEVE (AND [IN-SUIT-LED (CARD-OF ME)] [HIGH (CARD-OF ME)]))
```

DERIV8 operationalizes "get void in suit S0" as "keep playing cards in suit S0" by applying the same set depletion strategy used in DERIV3 (§2.5).

```
DERIV8 #STEPS 14 --DOMAIN: HEARTS
      --PROBLEM: (ACHIEVE (VOID ME S0))
      --SOLUTION: (UNTIL (VOID ME S0) (ACHIEVE (PLAY-SUIT ME S0)))
```

DERIV9 operationalizes the predicate "player P0 is void in suit S0" probabilistically as a decreasing function of the number of cards still out in suit S0. This is accomplished by applying a general formula for the probability that two sets are disjoint (§2.3), and then analyzing the functional dependence of this formula on the parameter S0 (§2.8).

```
DERIV9 #STEPS 59 --DOMAIN: HEARTS
      --PROBLEM: (EVAL (VOID P0 S0))
      --SOLUTION: (FUNCTION-OF (#CARDS-OUT-IN-SUIT S0) DECREASING)
```

DERIV10 uses historical reasoning (§2.4) to derive a procedure for counting the number of cards out in suit S0.

```
DERIV10 #STEPS 31 --DOMAIN: HEARTS
        --PROBLEM: (EVAL (#CARDS-OUT-IN-SUIT S0))
        --SOLUTION:
          (EVAL (- (- 13
                      (# (SET-OF X1 (CARDS-IN-SUIT S0)
                                    (BEFORE (CURRENT ROUND-IN-PROGRESS)
                                            (HAS ME X1)))))
                   (# (SET-OF X1 (CARDS-IN-SUIT S0)
                                 (WAS-DURING (CURRENT ROUND-IN-PROGRESS)
                                             (EXISTS P3 (OPPONENTS ME)
                                                     (PLAY P3 X1)))))))
```

DERIV11 operationalizes the predicate "player P0 is void in suit S0" as "player P0's card is not in the suit led" by applying a general strategy for finding a sufficient condition (§2.6).

```
DERIV11 #STEPS 18 --DOMAIN: HEARTS
        --PROBLEM: (EVAL (VOID P0 S0))
        --SOLUTION:
          (EVAL (AND [BREAKING-SUIT P0]
                     [= (SUIT-LED) S0]
                     [FOLLOWING P0]))
```

DERIV12 uses historical reasoning (§2.4) to operationalize the predicate "player P0 is void in suit S0" as "player P0 was void earlier in the round."

```
DERIV12 #STEPS 13 --DOMAIN: HEARTS
        --PROBLEM: (EVAL (VOID P0 S0))
        --SOLUTION:
          (EVAL (WAS-DURING (CURRENT ROUND-IN-PROGRESS) (VOID P0 S0)))
```

DERIV13 operationalizes the task of composing a *cantus firmus* (musical tone sequence) satisfying certain constraints as a heuristic search procedure (§3) incorporating the constraints as generators, tests, and precomputed data structures.

```
DERIV13 #STEPS 116 --DOMAIN: MUSIC
        --PROBLEM: (ACHIEVE (LEGAL-CANTUS! (CANTUS)))
        --SOLUTION: heuristic search in space of tone sequences
```

DERIV14 reformulates the constraint that the climax tone of the cantus should only occur once in it as a goal to be achieved over time (§2.11), and operationalizes this goal in a way that doesn't require selecting the climax tone before generating the cantus.

```
DERIV14 #STEPS 87 --DOMAIN: MUSIC --PROBLEM:
        (ACHIEVE (= (#OCCURRENCES (CLIMAX CANTUS) CANTUS) 1))
       --SOLUTION:
        (FORALL T1 (LB:UB 0 (- (# CANTUS) 1))
                (OR [AND [= (#OCCURRENCES (CLIMAX (CANTUS-1 T1))
                                          (CANTUS-1 T1))
                            1]
                         [LOWER (NOTE (+ T1 1))
                                (CLIMAX (CANTUS-1 T1))]]
                    [HIGHER (NOTE (+ T1 1)) (CLIMAX (CANTUS-1 T1))]))
```

1.2.3.2 Selection Criteria

An important criterion in selecting advice to operationalize was to choose examples that were challenging but could be solved by means of general principles, without sneaking in the solution as part of the input. For example, "lead a safe suit" was rejected because the only obvious definitions for "safe" were either already operational or else too vague to formalize.

Diversity was also a consideration in choosing examples. For example, three quite different approaches were applied to the problem of deciding whether an opponent is void, as shown in DERIV9 (§2.3), DERIV11 (§2.6.2), and DERIV12 (§2.4.2).

Another motivation was to encode knowledge about AI methods. The most sophisticated method encoded was the heuristic search method (HSM). In DERIV2, the advice "avoid taking points" was operationalized as a heuristic search problem by mechanically deriving an appropriate state space and search control heuristics from the representation of the advice (§3).

The goal of the research was to capture *general* knowledge about operationalization rather than achieve good task performance at the cost of *ad hoc* methods. Pursuing this goal did not require getting FOO to play Hearts, and in fact it doesn't.

Because of this emphasis on generality, only a few examples were treated relative to the range of potentially useful advice and the variety of operationalization methods applicable to it. FOO's collection of methods is a small and arbitrary sampling, neither complete nor representative. As an exploration of knowledge used in operationalization, this work is a beginning step in a large space.

1.2.4. Testing Generality

To test the generality of the approach, a music generation problem was selected as a second task and operationalized using FOO in the interactive mode described above, adding rules as necessary. The problem was inspired by an existing program for composing *cantus firmus* [Meehan 72]; the program incorporated aesthetic constraints from a music text [Salzer 69]. The task used to test FOO's generality was based on a subset of these constraints. In DERIV13, this task was operationalized as a heuristic search problem using many of the same rules used to operationalize "avoid taking points" in DERIV2. In DERIV14, an alternative operationalization was derived for one of the task constraints.

1.2.5. Testing Operationality

A potential problem with this method of investigation was the use of a subjective criterion for deciding when to terminate the interactive operationalization process and accept the resulting expression as *operational*. To address this problem, the operationality of each derived expression was analyzed by a simple procedure, as shown in Appendix E. Instead of returning a simple yes-or-no result, the procedure finds a set of conditions under which the expression will be operational. Making a yes-or-no decision would require the ability to predict what information will be known when an expression is evaluated (§8.1.1). The issue is further complicated by the fact that an operationalization can be better or worse depending on such factors as how often it can be applied, how much it costs, and how often it produces the desired result.

As the operationality analysis shows, some of the expressions were only partly operational or assumed the solution of other operationalization problems. For example, the plan generated in DERIV5 for flushing the Queen of spades without taking it is: "keep leading safe spades until the Queen is played." The success of this plan depends on getting and keeping the lead, and having enough spades to last until the Queen is flushed. Deciding whether the preconditions for success are satisfied is difficult. For example, the distribution of spades is unknown, so it's uncertain how many are "enough." Moreover, the plan may succeed even when the preconditions are not satisfied, for example if other players help out by also leading spades. The problem of getting the lead is addressed in DERIV7, but none of the derivations solve the problems of keeping the lead or deciding whether one has enough spades.

The plan produced in DERIV5 is partly operational insofar as it will sometimes be executable and when executed will sometimes succeed in flushing the Queen. A proposed approach for improving such a partly operational plan is to use it until it fails to work, diagnose the problem that caused the plan to fail, and modify the plan to prevent that problem from recurring [Hayes-Roth 81a]. DERIV5 actually illustrates the last phase of this approach, since it consists of modifying a plan for flushing the Queen (derived in DERIV3) so as to avoid taking the Queen oneself.

1.2.6. Organization of Results

The results of the investigation described above are organized according to the kinds of knowledge used in operationalization. (§2) presents general operationalization ideas underlying several derivations, and (§3) describes FOO's knowledge about the heuristic search method. (§4) discusses the crucial process of problem reformulation, which figures throughout the derivations. Typically only one or two of the rules used in a derivation express general ideas about operationalization; the rest of the derivation consists of the analysis required to apply these general ideas to the problem at hand. (§5) describes the methods used to perform this analysis. (§6) tells how task domain knowledge is used in operationalization. These chapters draw support and examples from the derivations generated using FOO in the interactive mode described above.

1.2.7. A Problem Space for Operationalization

The problem space in which FOO works can now be described more precisely. A *problem* consists of a piece of advice to be operationalized, encoded as an S-expression in a LISP-like language [McCarthy 63] described in detail in (§1.3.1). This language uses prefix notation: an expression (f e_1 ... e_n) denotes the result of applying the function f to the arguments e_1 ... e_n, where f, e_1, ..., e_n may themselves be expressions. An expression of the form (achieve P) represents the problem "find an operational procedure for achieving P." An expression of the form (eval e) represents the problem "find an operational procedure for evaluating e." Thus the expression (ACHIEVE (VOID ME S0)) represents the problem of *getting* void in suit S0, while (EVAL (VOID P0 S0)) represents the problem of *deciding* whether player P0 is void in suit S0.

A *solution* to such a problem is an expression that the assumed runtime interpreter can interpret to produce the desired result. For example, (UNTIL (VOID ME S0) (PLAY-SUIT ME S0)) is a plan for getting void in suit S0 by successively playing cards in that suit, while the expression (FUNCTION-OF (#CARDS-OUT-IN-SUIT S0) DECREASING) describes a probabilistic method for guessing whether an opponent is void in suit S0; it characterizes the probability of a void as a decreasing function of the number of cards still out in suit S0. This characterization can be used to order the suits according to the relative probabilities of an opponent being void in them.

A *solution path* or *derivation* in the problem space is a series of expressions leading from a problem to a solution, where each expression is the result of applying some transformation to the previous one. Certain (overlapping) classes of transformations are worth identifying:

1. A *valid* transformation preserves semantic equivalence, *i.e.*, transforms an expression to another with the same meaning.
2. A *sound* transformation changes a logical condition to an equivalent or stronger one, *i.e.*, never transforms falsehood into truth. All valid transformations are sound.
3. A *useful* transformation changes an expression into one that is closer to being operational. A transformation may be useful but not valid or sound, and *vice versa*. The problem of predicting whether a transformation will prove useful is beyond the scope of the dissertation but must be addressed as part of building a problem-solver for this space (§7).

An *operator* in the problem space is a rule for transforming an expression. A rule has the form

RULEnnn: <lhs> -> <rhs> if <condition>

The <lhs> and <rhs> are patterns that can contain pattern-match variables and embedded procedure calls. The <condition> is a predicate that can refer to the variables. To apply a rule to an expression, FOO matches the <lhs> of the rule to the expression, binds the variables in the <lhs> to the corresponding components of the expression, verifies the <condition>, replaces variables in the <rhs> with their bindings from the match, executes any procedure calls embedded in the <rhs>, and returns the resulting expression. If the rule was applied to a sub-expression of an overall problem, FOO transforms the problem by replacing the sub-expression with the result of applying the rule to it. Typically the existence of a match and the satisfaction of the <condition> guarantee that applying the rule will produce a valid or sound (but not necessarily useful) transformation.

For example, one of the rules in FOO is

RULE43: (R (f e_1 ... e_n) (f e_1' ... e_n')) -> (and [= e_1 e_1'] ... [= e_n e_n']) if R is reflexive

For some rules, verifying the <condition> or evaluating the <rhs> involves establishing a subproblem and applying other rules to it. Thus a *problem state* in the operationalization process may contain a stack of expressions representing subproblems (§5.9.3). In addition, the problem state may include various global assumptions and variable bindings used to facilitate communication between different problems (§5.4).

The rules shown in later examples are expressed in an informal notation that sacrifices some precision in favor of readability. For example, a rule containing the pattern (P ... e ...) in its <lhs> is ambiguous in that it doesn't specify whether e can be nested arbitrarily deeply or must be a top-level argument of P. If the rule is applied to an expression containing more than one sub-expression that matches e, this informal notation doesn't specify whether to replace one, some, or all of the sub-expressions matching e. Appendix B lists FOO's transformation rules in a precise, machine-generated format and explains in more detail how they are applied.

Although the rules are general, they can access a *domain-specific knowledge base*. Domain knowledge is stored in various forms. For example, the fact that a domain-specific concept like VOID is an instance of a general concept like PREDICATE is encoded in an is-a hierarchy. This enables FOO to apply general rules to domain-specific expressions. Concept definitions are encoded as lambda expressions in the same language used to encode problems and solutions. This enables FOO to apply knowledge about a concept used in a problem simply by replacing the concept with its definition, provided it knows the concept's definition. (FOO doesn't have a definition for every concept it knows about.) For example, the definition "being *void* in a suit means having no cards in that suit" is encoded as

```
VOID = (LAMBDA (P S)
          (NOT (EXISTS C1 (CARDS-IN-HAND P) (IN-SUIT C1 S))))
```

The variable names P, S, and C1 have no inherent significance, *i.e.*, variables are not typed.

A definition may itself refer to domain-specific concepts whose definitions refer to lower-level concepts, and so forth. For example, the concept of the cards held by a player is defined by

```
CARDS-IN-HAND = (LAMBDA (P) (SET-OF C1 (CARDS) (HAS P C1)))
```

The derivation of a plan for getting void (DERIV8) involves elaborating the concept of being void in terms of the concept of a card being at a location. This allows the application of the general knowledge that the way to cause an object to stop being at a location is to move it, and the domain-specific knowledge that in Hearts a legal way to move a card is to play it.

1.3. Representation of Conceptual Knowledge

The concepts used in FOO can be classified into two broad categories: *general* and *domain-specific*. Most of FOO's knowledge about domain-specific concepts is encoded in the form of *concept definitions* in a language described in (§1.3.1). For example, the fact that "taking points" means "taking a point card" is encoded as

```
TAKE-POINTS = (LAMBDA (P) (FOR-SOME C (POINT-CARDS) (TAKE P C)))
```

Most of FOO's knowledge about the general concepts used in rules is encoded in an *is-a hierarchy* whose terminal nodes include some domain-specific concepts. Thus the fact that "taking points" is an action is encoded as

```
TAKE-POINTS is-a ACTION
```

A few other relations are encoded in a *semantic network*. One fact encoded in the network is

```
OPPOSITE of HIGH is LOW
```

The is-a hierarchy and semantic network are described in (§1.3.2). Finally, a few facts about the domain are encoded as "*fact rules*," presented in (§1.3.3). For example, the fact that there are 13 cards in each suit is encoded as

RULE376: (# (cards-in-suit s)) -> 13

The various ways in which FOO uses *conceptual* knowledge are discussed in (§6).

The rest of (§1.3) describes FOO's knowledge representation in more detail; the reader may wish to skip to (§1.4), where the discussion of operationalization resumes.

1.3.1. A Language for Representing Concepts

FOO uses a LISP-like language to represent concept definitions, problems, and solutions. The representation is *applicative* insofar as assignment statements and goto's are not allowed, but certain constructs depend implicitly on the current state of the task environment. For example, `(LOC C)` is a *fluent* -- a function of time whose value is the location of card `C` at time t. Similarly, the expression `(CURRENT TRICK)` denotes the instantiation of the `TRICK` scenario current at time t.

Atomic symbols denote *variables*, *constants*, or *functions*. FOO uses three kinds of variables:

1. *Lambda variables* are used as formal arguments in concept definitions.
2. *Global variables* are generated by rules to represent unknowns in equations.
3. *Quantifier variables* are used in quantified expressions of the form $(Q\ x\ S\ e_x)$.

To prevent name conflicts, quantifier variables and global variables are given unique names like `C1` and `P4` whenever they're generated. A variable can be bound to a value at most once; FOO's representation does not have standard programming language variables.

Atomic symbols other than variables are treated as unevaluated *constants, e.g.*, ME, QS, 1. This feature differs from LISP semantics, which uses the QUOTE construct to represent non-numeric constants. FOO has no QUOTE construct. Note that the same symbol can be a lambda variable when used inside a concept definition and a constant elsewhere. For example, S is sometimes used as a lambda variable denoting a set, but otherwise denotes the spade suit.

An atomic symbol can also denote a *function*. In the implementation, atoms are marked so the system can distinguish among variables, constants, and functions. Concepts are represented in FOO as LISP functions, using LISP's built-in mechanisms for encoding and editing function definitions. (Some partial definitions are encoded as rules (§5.5).)

Non-atomic expressions follow LISP's prefix notation. There are four kinds of list expressions:

1. *Lambda expressions* have the form (lambda (x_1 ... x_n) <expression>).
2. *Quantified expressions* have the form (Q x S e_x), where variable x ranges over set S.
3. *Extensional collections* have the form (L e_1 ... e_n), where L is a list-ary function.
4. *Function applications* have the form (f e_1 ... e_n), where f is an n-ary function.

1.3.1.1 Lambda Variables versus Slots

The lambda variables x_1 ... x_n of a function definition f = (lambda (x_1 ... x_n) <body>) correspond to the "slots" or "facets" of the concept f, but FOO's representation has no explicit data structure corresponding to the type constraints, inheritance information, and other knowledge often attached to such components; they simply weren't needed. There appear to be three reasons for this.

First, the usual purpose of such information in an AI system is to help *disambiguate, complete*, or *instantiate* a concept, but the problems given to FOO and the concept definitions used in solving them are encoded to begin with as *unambiguous, well-formed* expressions. Since FOO receives well-formed input, no instantiation or disambiguation is needed to interpret it. Significantly, the treatment of the heuristic search method in FOO does involve instantiating a schema, and FOO has knowledge about individual slots of that schema, encoded as rules (§3.2).

A second factor obviating the need for slot information is the *strong typing* inherent in FOO's representation. Specifically, the only variables used in concept definitions are quantifier variables, and all quantification is restricted to the form (Q x S P_x) where S is the scope of variable x.

Finally, I've avoided using slots when functions will do, an approach inspired by discussions with

Stan Rosenschein (who bears no responsibility for any bugs in FOO's representation). For example, a typical AI system might encode the fact that a card has suit, rank, and possibly points as a schema for CARD with slots for SUIT, RANK, and POINTS. FOO *uses functions instead of annotated slots.* For example, the function SUIT-OF returns the suit of a given card, and the function POINTS returns the number of points a given card has. A card could be specified by its suit and rank by defining a function

```
CARD = (LAMBDA (S R)
           (THE C (CARDS)
              (AND [= (SUIT-OF C) S] [= (RANK-OF C) R])))
```

1.3.1.2 General versus Specific Concepts

Classifying concepts as general or domain-specific is a bit simplistic. The concepts in FOO span a spectrum of generality, with Hearts-specific concepts like TAKE-POINTS at one extreme, and immutable mathematical concepts like equality at the other. In between are card game concepts like PLAY, and basic concepts like MOVE that may be modelled differently in different domains. For example, rate of motion is irrelevant in Hearts but important in physics.

Some of the concepts in FOO are extremely specific, like #CARDS-OUT-IN-SUIT and LEAD-SAFE-SPADE, and the suspicious reader may consider them evidence that I sneaked into the knowledge base the solutions to the problems solved using FOO. These concepts are simply the result of giving names to expressions that arose naturally in the course of operationalization. For example, (#CARDS-OUT-IN-SUIT SUIT) is a convenient name for the expression (# (FILTER (CARDS-IN-SUIT SUIT) OUT)) synthesized by one of the methods used to decide if an opponent is void (§2.3). Similarly, (LEAD-SAFE-SPADE P) is the convenient name given to an expression synthesized in the course of operationalizing the advice "flush the Queen without taking it" (§2.10):

```
(AND [LEADING P] [SPADE! (CARD-OF P)] [HIGHER QS (CARD-OF P)])
```

This particular "definition of convenience" can be criticized on the grounds that it doesn't explicitly mention playing a card. However, omitting it from the knowledge base would not hinder the synthesis of an operational plan to satisfy the advice, only its expression in a more compact form.

1.3.1.3 Concepts Corresponding to Different Parts of Speech

Describing FOO's concept representation as a "LISP-like language" (§1.3.1) fails to specify how to encode a given concept. This section tries to give a more concrete idea of how knowledge is encoded in practice by showing how concepts corresponding to the same parts of speech are encoded in similar forms.

Nouns are encoded in FOO in several ways.

1. *Constants* like `QS` and `ME` denote *named objects* like queen of spades, system-as-player.
2. (the x S P_x) denotes a *singular definite* noun -- *the* (unique) x in S satisfying P. Thus `(OWNER-OF C) = (THE P (PLAYERS) (HAS P C))` denotes *the* player holding card `C`.
3. (set-of x S P_x) or (filter S P) denotes a *plural definite* noun -- *those* x in S satisfying P: `(CARDS-IN-POT) = (SET-OF C (CARDS) (AT C POT))` denotes all the cards in the pot.
4. A *quantifier variable* x in (Q x S P_x) denotes a *singular indefinite* noun: `(OUT C) = (EXISTS P (OPPONENTS ME) (HAS P C))` means "a card is out if *an* opponent has it."

Adjectives (predicates and noun modifiers) correspond to various constructs in FOO.

1. *Unary predicates* denote absolute adjectives like `HIGH`.
2. *Binary relations* denote *comparatives* like `HIGHER`.
3. (lambda (S) (the x S (forall y S (not (R y x))))) denotes *superlatives* like `HIGHEST`.

Note that this representation scheme covers predicates and their intensions -- "the *set of* x satisfying P," or "*the unique* x satisfying P" -- but not indefinite nouns like "*an* x satisfying P."

Action verbs are encoded in FOO as predicates. An action (A e_1 ... e_n) can be defined as a fluent (function of time) in more than one way, *e.g.*:

(lambda (t) <specified action starts at time t>)

(lambda (t_1 t_2) <specified action starts at time t_1 and ends at time t_2>)

In practice, expressions denoting actions are manipulated without making the time parameter explicit. An example of an *action verb* definition is `PLAY = (LAMBDA (P C) (MOVE C (HAND P) POT))`, which denotes "playing a card means moving it from one's hand to the pot." It is unnecessary to make `P` an explicit argument of `MOVE`, since in the game of Hearts the origin and destination of a motion uniquely determine its agent. The lowest-level action is "fluent `F` becomes `C`," encoded as `(BECOME F C)`. If time is modelled discretely and represented by the integers, "F becomes C at time i" can be interpreted as "F equals C at time i+1 but not at time i." This enables the following definitions:

```
BECOME =
  (LAMBDA (F C)
     [LAMBDA (I) (AND [NOT (= (F I) C)] [= (F (+ I 1)) C])])

MOVE =
  (LAMBDA (OBJ ORIGIN DESTINATION)
     (AND [= (LOC OBJ) ORIGIN] [BECOME (LOC OBJ) DESTINATION]))

CAUSE = (LAMBDA (P) (BECOME P T))

UNDO = (LAMBDA (P) (BECOME P NIL))
```

Verbs of being are also encoded as predicates. Thus (HAS P C) = (AT C (HAND P)) means "a player *has* a card if the card is located at the player's hand."

Adverbs (action modifiers) are encoded as *predicates.*

1. The condition C in an action verb expression (while C A) restricts the action A. This is *not* the standard programming language sense of "while." (LEAD P C) = (WHILE (LEADING P) (PLAY P C)) denotes "lead card C means play C while leading."
2. A proposition conjoined with an action modifies it, since actions are predicates. Thus (PLAY-SPADE P) = (AND [PLAY P ?C] [SPADE! ?C]) means "play a spade." In this lazy notation, the variable ?C is implicitly existentially quantified.
3. (Q x S A_x) denotes an *indeterminate, repeated,* or *choice-dependent* action. For example, (FOR-SOME C (POINT-CARDS) (TAKE P C))) denotes "take a point card"; (EACH C (CARDS-PLAYED) (TAKE P C)) denotes "take all the cards played"; and (CHOOSE (CARD-OF P) (LEGALCARDS P) (PLAY P (CARD-OF P))) denotes "choose a legal card and play it."
4. (scenario A_1 ... A_n) denotes a sequence of actions. For example, a trick is defined as

   ```
   (SCENARIO (EACH P (PLAYERS) (PLAY-CARD P))
             (TAKE-TRICK (TRICK-WINNER)))
   ```
5. (until C A) denotes an iterated action. For example, a plan for flushing the queen is

   ```
   (UNTIL (PLAYED! QS) (ACHIEVE (LEAD-SPADE ME))).
   ```
6. (was-during S A) and (before S A) denote *past tense:* "P *took* C *during the round*" is encoded as (WAS-DURING (CURRENT ROUND-IN-PROGRESS) (TAKE P C)), and "C was out when the round started" as (BEFORE (CURRENT ROUND-IN-PROGRESS) (OUT C)).

Attributes are encoded as *functions.* Thus (# S) denotes the *size* of set S, and (HAND P) denotes "player P's hand" (a location, not the set of cards there). Some attributes of sets are defined by *projecting* attributes of the set's members. For example, "the players' hands" is encoded as (HANDS (PLAYERS)), where HANDS = (LAMBDA (S) (PROJECT HAND S)).

Prepositional phrases are encoded as *relations*. For example, "the cards *in the pot*" is encoded as (SET-OF C (CARDS) (AT C POT)), and "take points *during the current trick*" is encoded as (DURING (CURRENT TRICK) (TAKE-POINTS ME)). Note that the predicate DURING is actually a functional, since its arguments are actions (predicates). Suppose actions are modelled as predicates of the form

(lambda (t_1 t_2) <action starts at time t_1 and ends at time t_2>)

Then "A occurs during S" means "if S occurs over some time interval, A occurs in a sub-interval":

```
DURING =
  (LAMBDA (S A)
     [FORALL T1 (TIME)
        (FORALL T2 (TIME)
           (=> [S T1 T2]
               [EXISTS T3 (TIME)
                  (EXISTS T4 (TIME)
                     (AND [A T3 T4] [=< T1 T3] [=< T4 T2]))])])
```

This definition is shown here only to demonstrate that DURING can be assigned a precise meaning. With its quadruply nested quantification, it would be rather unwieldy to reason about, and it is not included in FOO's knowledge base.

Connectives are encoded using standard logical operators:

1. (and C_1 ... C_n) is true when all of C_1 ... C_n are true and false otherwise.
2. (or C_1 ... C_n) is true when any of C_1 ... C_n are true and false otherwise.
3. (not C) is true when C is false, and *vice versa.*
4. (=> C C') is true except when C is true and C' is false.
5. (if C e e') denotes "if C then e otherwise e'," where e need not be boolean.

The concept definitions used in the derivations are listed in Appendix C.3.

1.3.2. Semantic Relations

Some knowledge in FOO is encoded as *semantic relations* between concepts. The most common of these is the is-a relation. Since this relation is binary and anti-symmetric, it can be encoded as a lattice (acyclic graph) with unlabelled arcs. FOO also uses a few other binary semantic relations. These are encoded in a graph whose arcs are labelled with the relation name.

1.3.2.1 Is-a Hierarchy

The concepts used in FOO are organized into an is-a hierarchy. The hierarchy is *tangled* in that a concept may have more than one immediate generalization. For example, the SUBSET relation is both REFLEXIVE and TRANSITIVE. An alternative view of the is-a hierarchy is that it compactly encodes concept *features* by exploiting subsumption relations between features.

The hierarchy contains both general and domain-specific concepts. For example,

```
TAKE-POINTS is-a ACTION is-a PREDICATE is-a FUNCTION is-a ANY-CONCEPT
```

The same mechanism is used to distinguish Hearts concepts from music concepts; thus TAKE-POINTS is-a HEARTS and EXTEND is-a MUSIC. This helps FOO ignore irrelevant concepts when searching the knowledge base for concepts of a given form (§6.5). Note the loose use of "is-a"; more accurate would be "has-feature" or "is-related-in-unspecified-way-to."

FOO's is-a hierachy is shown in Appendix C.1.

1.3.2.2 Semantic Network

Certain semantic relations between concepts are encoded using LISP's property-list (attribute-value pair) mechanism. Since they are all binary relations and single-valued functions of their first argument, they can be expressed as facts of the form "the <attribute> of <concept> is <value>" and encoded by putting the ordered pair (<attribute> <value>) on the property list of <concept>.

This representation is useful when FOO lacks a general definition for <attribute>. For example, it is difficult to define "opposite," but it is important for FOO to know that the opposite of "high" is "low." This fact is represented as the semantic relation

```
OPPOSITE of HIGH is LOW
```

Some other relations between adjectives include:

```
PREDICATE of HIGHER is HIGH
COMPARATIVE of HIGHEST is HIGHER
SUPERLATIVE of HIGHER is HIGHEST
```

The IDENTITY property gives the identity element of a specified operator:

```
IDENTITY of AND is T
IDENTITY of D* is INCREASING
```

Finally, the ADD property encodes information about homomorphisms. A homomorphism is a function f such that (f (+ x y)) = (+' (f x) (f y)) where + and +' represent addition-like operators in the domain and range of f, respectively. Information about homomorphisms could be straightforwardly encoded as a ternary relation (H f + +'). Rather than implementing a special mechanism to handle ternary relations, I exploited the fact that f uniquely determined +' for the few instances of the relation H used in the derivations. The ADD property maps f to +', *e.g.*, the homomorphic identity (in e (union S1 ... Sn)) = (or [in e S1] ... [in e Sn]) is encoded as

```
ADD of IN is OR
```

The semantic relations encoded in FOO are listed in Appendix C.2.

1.3.3. Domain-specific Rules

FOO uses a few domain-specific "fact rules" to simulate unimplemented mechanisms. Typically, a fact about a concept is represented as a rule in the absence of a definition for the concept. For example, the fact that the number of cards in each suit is 13 is represented as

RULE376: (# (cards-in-suit s)) -> 13

This is easier than representing the concept of set cardinality (denoted by "#") as a definition of the form

```
# = (LAMBDA (S) <what goes here?>)
```

Moreover, it is not clear that such a definition would provide an efficient way to compute the number of cards in a suit.

1.3.3.1 Simulated Types

Some fact rules simulate a type mechanism:

RULE62: (in QS (cards)) -> T -- *QS is a card*

RULE173: (in ME (players)) -> T -- *ME is a player*

RULE197: (domain suit-of) -> (cards) -- *argument of suit-of is a card*

RULE363: (subset (cards-played) (cards)) -> T -- *(cards-played) is a set of cards*

RULE365: (in (leader) (players)) -> T -- *(leader) is a player*

1.3.3.2 Simulated Partitioning

Some rules simulate a NETL-like scheme [Fahlman 79] for encoding information about mutually exclusive subsets or cases:

RULE163: (range loc) -> (partition [hands (players)] [piles (players)] [set deck pot hole])

RULE299: (not (or higher =)) -> lower -- `HIGHER` *and* `LOWER` *are domain-specific*

1.3.3.3 Simulated Property Retrieval

Other rules simulate a mechanism for retrieving properties of objects:

RULE2: (suit-of QS) -> S

RULE376: (# (cards-in-suit s)) -> 13

1.3.3.4 Simulated Deductions

Finally, several rules represent facts FOO already has enough knowledge to prove, or might be able to deduce from a complete description of the task:

RULE1: (has (owner-of c) c) -> T

RULE61: (actions-of p) -> (play-card p) if (trick-in-progress) -- *actions-of = legal moves*

RULE146: (= (player-of c) p) -> (= (card-of p) c)

RULE180: (at card deck) -> nil if (round-progressing)

RULE228: (legal p c) -> T if (= c (card-of p))

RULE279: (change cantus-1) -> (list (next note))

RULE357: (before (current round-in-progress) (at card (pile p))) -> nil

RULE372: (before (current round-in-progress) (in-pot c)) -> nil

RULE373: (at c hole) -> nil by assumption -- *instance of "assume C is false if unlikely"*

These 19 rules are the only fact rules used in the derivations. The rest of FOO's rules are general.

1.4. Knowledge Used in Operationalization

A view of operationalization that emerges from the derivations can be summarized (with apologies to Edison) by the equation

Operationalization = 2% Inspiration + 98% Perspiration

In this case

Inspiration = general ideas about operationalization

Perspiration = domain knowledge and analysis required to apply the general ideas

Some general operationalization methods used in the derivations include:

Reformulate a goal as an evaluable predicate on a choice variable. (§4.3.1.2)

Use heuristic search to evaluate a predicate on a sequence of choices. (§3)

Find an evaluable necessary or sufficient condition for an unevaluable predicate. (§2.6)

Evaluate a predicate on a set in terms of the probability that it's disjoint with another set. (§2.3)

Approximate an expression as an increasing or decreasing function of some quantity. (§2.8)

Use the value of a predicate from an earlier situation if it can't have changed since then. (§2.4)

1.5. Analysis Techniques

Applying these methods to a given problem typically requires analysis based on domain knowledge. A variety of analysis techniques (§5) are illustrated in the following excerpts from DERIV6, which also serve to introduce notation used in later examples. (The complete derivation is 43 steps long.) In this derivation, the advice "avoid taking points" is operationalized as "make my card low" by applying the general strategy

Reformulate a goal as an evaluable predicate on a choice variable.

The initial advice is encoded as

```
(AVOID (TAKE-POINTS ME) (TRICK))
```

This actually means "avoid taking points *during the trick*." The added specification simplifies the operationalization problem by disambiguating the advice, since "avoid taking points" is ambiguous with respect to the time interval over which the act of taking points is to be avoided.

The initial expression is non-operational for the assumed runtime mechanism because it's not expressed in terms of that mechanism's built-in capabilities, *i.e.*, it doesn't specify what card to play in a given situation in order to avoid taking points. To operationalize the advice, it is necessary to apply knowledge about the concepts in terms of which the advice is expressed. These concepts are defined as follows:

> *Avoid* an event over some period means try not to let the event occur during the period.
>
> *Take points* means take a point card.
>
> A *trick* is a scenario in which each player plays a card and then the winner takes them all.

These definitions are encoded in FOO as follows:

```
AVOID = (LAMBDA (EVENT PERIOD)
           (ACHIEVE (NOT (DURING PERIOD EVENT))))

TAKE-POINTS = (LAMBDA (PLAYER)
                 (FOR-SOME C (POINT-CARDS) (TAKE PLAYER C)))

TRICK = (LAMBDA ()
           (SCENARIO
              (EACH P (PLAYERS) (PLAY-CARD P))
              (TAKE-TRICK (TRICK-WINNER))))
```

1.5.1. Elaboration

First the problem is *elaborated* (§5.8.2) by expanding the definitions of AVOID and TRICK:

```
(AVOID (TAKE-POINTS ME) (TRICK))

6:2-3 --- [ELABORATE by RULE124] --->

(ACHIEVE (NOT (DURING (SCENARIO
                         (EACH P1 (PLAYERS) (PLAY-CARD P1))
                         (TAKE-TRICK (TRICK-WINNER)))
                      (TAKE-POINTS ME))))
```

The notation "6:2-3" refers to steps 2 through 3 in the derivation DERIV6. The notation "--- [ELABORATE by RULE124] --->" between the two expressions indicates that the second expression is the result of elaborating the first, and that this transformation is effected by RULE124, shown below:

> RULE124: $(f\, e_1 \ldots e_n) \rightarrow e'$, where f is defined as (lambda $(x_1 \ldots x_n)$ e)
> and e' is the result of substituting $e_1 \ldots e_n$ for $x_1 \ldots x_n$ throughout e

1.5.2. Case Analysis

The resulting expression is non-operational since it depends on the outcome of the trick, which is generally unforeseeable at the time player ME (FOO's card-playing *persona*) must choose a card consistent with the advice. Thus the expression must be analyzed to determine which properties of the card player ME plays may enable it to avoid taking points. *Case analysis* (§5.7.1) shows that player ME can only take points by winning the trick:

```
(ACHIEVE (NOT (DURING (SCENARIO
                         (EACH P1 (PLAYERS) (PLAY-CARD P1))
                         (TAKE-TRICK (TRICK-WINNER)))
                      (TAKE-POINTS ME)))))

6:4            --- [DISTRIBUTE by RULE284] --->

(ACHIEVE (NOT (OR [DURING (EACH P1 (PLAYERS) (PLAY-CARD P1))
                          (TAKE-POINTS ME)]
                  [DURING (TAKE-TRICK (TRICK-WINNER))
                          (TAKE-POINTS ME)])))
```

This transformation is performed by

RULE284: $(f ... (+ e_1 ... e_n) ...) \rightarrow (+' (f ... e_1 ...) ... (f ... e_n ...))$
where + is an addition-like operator over the domain of f, +' is the corresponding operator over the range of f, and (lambda (x) (f ... x ...)) is a homomorphism[4] with respect to + and +'

Here f = DURING. FOO identifies g = (LAMBDA (X) (DURING X (TAKE-POINTS ME))) as a homomorphism and + = SCENARIO as an addition-like operator by searching through an is-a hierarchy (§1.3.2.1) and finding paths from DURING to HOMOMORPHISM and from SCENARIO to JOIN. It retrieves +' = OR from a semantic network containing the attribute-value relation ADD of SCENARIO is OR (§1.3.2.2).

Note that RULE284 is applied to the *sub-expression* (DURING ...) of the overall problem

```
(ACHIEVE (NOT (DURING (SCENARIO
                        (EACH P1 (PLAYERS) (PLAY-CARD P1))
                        (TAKE-TRICK (TRICK-WINNER)))
                      (TAKE-POINTS ME)))))
```

This is indicated by indenting the arrow representing step 6:4 so that it lines up under the beginning of the transformed sub-expression. The same convention is used throughout the dissertation where feasible. For simplicity, the context surrounding a transformed sub-expression is often abbreviated or omitted when displaying the transformation.

[4]A function g is a homomorphism with respect to additive operators + and +' if $g(x+y) = g(x) +' g(y)$.

1.5.3. Intersection Search

At this point, "taking points" has been split into two cases: taking points during the part of the trick when cards are played, and taking points when the winner takes the trick. The first case is eliminated based on the fact that a TAKE-POINTS event can't occur during a sequence consisting exclusively of PLAY-CARD events:

```
(DURING (EACH P1 (PLAYERS) (PLAY-CARD P1)) (TAKE-POINTS ME))
6:5 --- [COMPUTE by RULE57] --->
NIL
```

This is accomplished by

RULE57: (during s e) -> nil if s and e have no common sub-events

FOO tests RULE57's condition by performing an *intersection search* through the knowledge base of defined concepts (§5.6) for a common sub-event of s and e. Here e = (TAKE-POINTS ME) and s = (EACH P1 (PLAYERS) (PLAY-CARD P1)).

The sub-events of an event include the action concepts used to define it, the lower-level concepts in terms of which *those* actions are defined, and so forth, down to but not including primitive concepts like MOVE. In the current example, s = (EACH P1 (PLAYERS) (PLAY-CARD P1)), so one sub-event of s is the action concept PLAY-CARD. This concept is defined by

```
PLAY-CARD = (LAMBDA (P) (CHOOSE C1 (LEGALCARDS P) (PLAY P C1)))
```

That is, PLAY-CARD refers to the action of choosing a legal card and playing it. Thus PLAY is a sub-event of PLAY-CA^D and hence of s. Similarly, a sub-event of e = (TAKE-POINTS ME) is TAKE-POINTS, defined by

```
TAKE-POINTS = (LAMBDA (P) (FOR-SOME C1 (POINT-CARDS) (TAKE P C1)))
```

This definition says that taking points means taking some point card. TAKE is a sub-event of TAKE-POINTS. Thus the sub-events of s are PLAY-CARD and PLAY, while the sub-events of e are TAKE-POINTS and TAKE. Since these two sets are disjoint, RULE57's condition is satisfied.

1.5.4. Partial Matching

The concepts TAKE-TRICK and TAKE-POINTS have the common sub-event TAKE. The problem is elaborated by plugging in their definitions and *restructured* (§5.10) by extracting quantifiers:

```
(DURING (TAKE-TRICK (TRICK-WINNER))
        (TAKE-POINTS ME))

6:7-10 --- [ELABORATE by RULE124, restructure] --->

(EXISTS C3 (CARDS-PLAYED)
   (EXISTS C1 (POINT-CARDS)
      (DURING (TAKE (TRICK-WINNER) C3)
              (TAKE ME C1))))
```

FOO has no definition for the DURING predicate, but knows it's reflexive. This guarantees that it is logically sound to *partial-match*(TAKE (TRICK-WINNER) C3) against (TAKE ME C1) (§5.3) using

RULE43: (R $(f\,e_1 \ldots e_n)$ $(f\,e_1' \ldots e_n')$) -> (and [= e_1 e_1'] ... [= e_n e_n']) if R is reflexive

RULE43 is sound because its right side logically implies its left side if R is reflexive. Here R = DURING and f = TAKE. Applying RULE43 reduces the expression to a conjunction of equalities:

```
(EXISTS C3 (CARDS-PLAYED)
   (EXISTS C1 (POINT-CARDS)
      (DURING (TAKE (TRICK-WINNER) C3)
              (TAKE ME C1))))

6:11  --- [REDUCE by RULE43] --->

(EXISTS C3 (CARDS-PLAYED)
   (EXISTS C1 (POINT-CARDS)
      (AND [= (TRICK-WINNER) ME]
           [= C3 C1])))
```

The quantifier variable C1 is eliminated by applying some *simplification* rules (§5.2):

```
(EXISTS C3 (CARDS-PLAYED)
   (EXISTS C1 (POINT-CARDS)
      (AND [= (TRICK-WINNER) ME]
           [= C3 C1])))

6:12-13 --- [SIMPLIFY by RULE59, RULE108] --->

(AND [= (TRICK-WINNER) ME]
     [EXISTS C3 (CARDS-PLAYED) (HAS-POINTS C3)])
```

1.5.5. Recognizing Known Concepts

The second conjunct is now *recognized* (§5.8.3) as a known concept:

```
(EXISTS C3 (CARDS-PLAYED) (HAS-POINTS C3))
6:21-22 --- [RECOGNIZE by RULE123] --->
(TRICK-HAS-POINTS)
```

This is accomplished by

RULE123: e -> (f e_1 ... e_n) if f is defined as (lambda (x_1 ... x_n) <body>)
and e is the result of substituting e_1 ... e_n for x_1 ... x_n throughout <body>

Recognizing a known concept could enable application of previously learned knowledge about it, *e.g.*, ways to predict the likelihood that a trick will have points (§6.3.1).

1.5.6. Approximating an Expression as a Function of Available Data

Taking points has now been narrowed down to winning a trick that has points. Some analysis of the definition of TRICK-WINNER leads to the expression

```
(ACHIEVE (NOT (AND [= (TRICK-WINNER) ME]
                   [TRICK-HAS-POINTS])))
6:23-42 --- [by analysis] --->
(ACHIEVE (=> [AND [IN-SUIT-LED (CARD-OF ME)]
                  [TRICK-HAS-POINTS]]
             [LOWER (CARD-OF ME)
                    (FIND-ELT (CARDS-PLAYED-IN-SUIT-LED))]))
```

This means "if you're following suit in a trick with points, try to underplay some other card played in the suit led." Since (CARDS-PLAYED-IN-SUIT-LED) is generally unknown when player ME chooses (CARD-OF ME), the expression is not operational. Dependence on the unknown set is eliminated by using the unary predicate LOW to *approximate* the binary relation LOWER (§2.8.7):

```
             (LOWER (CARD-OF ME)
                    (FIND-ELT (CARDS-PLAYED-IN-SUIT-LED)))
6:43         --- [REDUCE by RULE154] --->
             (LOW (CARD-OF ME))
```

This transformation is not logically sound, but is certainly useful. It is effected by

RULE154: (R e_1 e_2) -> (P e_1) where R is the comparative form of predicate P

Here R = LOWER. FOO finds P = LOW by retrieving the semantic relation

```
PREDICATE of LOWER is LOW
```

The resulting expression is a predicate on the choice variable (CARD-OF ME):

```
(ACHIEVE (=> [AND [IN-SUIT-LED (CARD-OF ME)]
                  [TRICK-HAS-POINTS]]
             [LOW (CARD-OF ME)]))
```

If LOW is treated as a fuzzy predicate [Zadeh 79], the expression is likewise a fuzzy predicate, and can be used to order potential candidates for (CARD-OF ME). The expression is evaluable if the runtime mechanism can evaluate (TRICK-HAS-POINTS). This is not possible in general since it is typically impossible to predict at card-choosing time whether the trick will have points. One way to get around this is to assume the trick will have points in the absence of knowledge to the contrary. This could be done by applying

RULE318: P -> (possible P), where (possible P) is true unless P is known to be false.

RULE318 is based on three-valued logic (§2.6.3). Note that (possible P) is evaluable even when P is not. Applying RULE318 produces the expression

```
(ACHIEVE (=> [AND [IN-SUIT-LED (CARD-OF ME)]
                  [POSSIBLE (TRICK-HAS-POINTS)]]
             [LOW (CARD-OF ME)]))
```

This expression represents the plan "If you're following suit in a trick that may have points, make your card low." This plan is an operational approximation of the advice "avoid taking points": it is not always effective, but it is always applicable (assuming that player ME has a low card, or that "low" is interpreted as "lowest available").

1.6. Operationalization Scenarios

The example just presented illustrates a common scenario in the operationalization process, shown in Figure 1-2: using *analysis techniques* (§5) and *domain knowledge* (§6), the expression is *reformulated* (§4) as a computable function of available information, such as choice variables and observable data.

A more complex operationalization scenario is shown in Figure 1-3: the expression is mapped to a known *operationalization method* (§2) whose application produces an operational expression, or something closer to it. For example, the advice "avoid taking points" can be mapped onto the

```
                    reformulate
expression ----------------------------> operational expression
```

Figure 1-2: Operationalization by reformulation

```
expression
    |
    | reformulate
    v             operationalization method
input for method ----------------------------> result of method
                                                    |
                                                    | reformulate
                                                    v
                                               operational expression
```

Figure 1-3: Operationalization by applying a method

heuristic search method (§3), producing a procedure that searches through the space of possible card sequences for the trick to see if any of them cause player ME to take points, *i.e.*, contain points and have player ME's card as the highest in the suit led.

The scenario depicted in Figure 1-3 can be elaborated by identifying some common stages in the application of an operationalization method:

1. *Reformulate* an expression until it's possible to
2. *Recognize* that the method is applicable and decide to apply it, so
3. *Reformulate* the expression to match the method problem statement and
4. *Fill in* additional information required by the method; then
5. *Refine* the instantiated method by applying additional domain knowledge.

At this point the instantiated method is ready for execution or application of other methods. Not all of these stages are explicit and separate every time a method is applied, but when they do occur, they tend to occur in the order described.

DERIV9 clearly illustrates these stages. In this derivation, the *disjoint subsets method* (§2.3) is used to solve the operationalization problem (EVAL (VOID P0 S0)) by deriving a formula to estimate the probability that a player P0 is void in a suit S0. The key idea of the method is to apply a general formula for the probability that two randomly selected subsets S and S' of a universe U will be disjoint. Since the formula depends only on the sizes of S, S', and U, this method can be useful for evaluating predicates on unknown sets.

The initial expression is *reformulated* by elaboration:

```
        (VOID PO SO)
9:1     --- [ELABORATE by RULE124] --->
        (NOT (EXISTS C1 (CARDS-IN-HAND PO) (IN-SUIT C1 SO)))
```

RULE189 *recognizes* this expression as a predicate on the unknown set (CARDS-IN-HAND PO) and consequently *decides to apply* the disjoint subsets method. This requires *reformulating* the expression in terms of disjoint subsets in order to match the method's problem statement. In particular, it is necessary to solve for S, S', and U. This involves considerable analysis:

```
        (NOT (EXISTS C1 (CARDS-IN-HAND PO) (IN-SUIT C1 SO)))
9:2-21 --- [by RULE189, analysis] --->
        (PR-DISJOINT (CARDS-IN-HAND PO)
                     (CARDS-IN-SUIT SO)
                     (COMMON-SUPERSET (CARDS-IN-HAND PO)
                                      (CARDS-IN-SUIT SO)))
```

This expression denotes the probability that the cards held by player PO and the cards in suit SO are disjoint subsets of an unspecified common superset. The next step is to *fill in* this information unspecified in the original problem but required by the method:

```
                     (COMMON-SUPERSET (CARDS-IN-HAND PO)
                                      (CARDS-IN-SUIT SO))
9:23-25              --- [by intersection search] --->
                     (CARDS)
```

The resulting solution is *refined* by considering only those cards satisfying some predicate satisfied by every element of (CARDS-IN-HAND PO). A knowlege base search finds the predicate OUT; a card is "out" if some opponent has it. Verifying that all cards held by an opponent PO are out involves some analysis:

```
        (PR-DISJOINT (CARDS-IN-HAND PO)
                     (CARDS-IN-SUIT SO)
                     (CARDS))
9:27-39 --- [REFINE by RULE200, analysis] --->
        (PR-DISJOINT (CARDS-IN-HAND PO)
                     (CARDS-OUT-IN-SUIT SO)
                     (CARDS-OUT))
```

The method is *applied* by using a formula for the probability that two randomly chosen subsets of a set are disjoint:

```
        (PR-DISJOINT (CARDS-IN-HAND P0)
                     (CARDS-OUT-IN-SUIT S0)
                     (CARDS-OUT))
    9:40-43 --- [APPLY by RULE124, analysis] --->
        (PR-DISJOINT-FORMULA (#CARDS-IN-HAND P0)
                             (#CARDS-OUT-IN-SUIT S0)
                             (#CARDS-OUT))
```

Given a procedure for computing PR-DISJOINT-FORMULA and methods for evaluating its three arguments, this expression is operational and can be evaluated at runtime.

An alternative to evaluating the expression is to analyze its *functional dependence* on one of its arguments (§2.8):

```
    (EVAL (PR-DISJOINT-FORMULA (#CARDS-IN-HAND P0)
                               (#CARDS-OUT-IN-SUIT S0)
                               (#CARDS-OUT)))
    9:44-57 --- [by RULE202, analysis] --->
    (FUNCTION-OF (#CARDS-OUT-IN-SUIT S0) DECREASING)
```

This expression means "a decreasing function of the number of cards out in suit S0." It can be used to order the suits according to the probability of a void in them, without actually computing the probabilities (§2.8.3). This approximation technique is especially useful for expressions that cost too much to evaluate or contain unknown terms.

Some of the stages in applying an operationalization method involve a considerable amount of logical analysis or "theorem-proving". For instance, *reformulating* "avoid taking points" as a heuristic search problem requires figuring out the sequence of choice points in a trick in order to formulate the search space. *Refining* the search involves finding tests that predict whether a partial path through this space can be extended into a scenario in which points are taken; this requires the ability to analyze logical relationships between partial and total paths. For example, if CARDS-PLAYED1 is a subsequence of the complete sequence (CARDS-PLAYED) of cards played in the trick, these two implications hold:

```
    (AND [IN (CARD-OF ME) CARDS-PLAYED1]
         [= (CARD-OF ME) (HIGHEST-IN-SUIT-LED (CARDS-PLAYED))]) =>
    (= (CARD-OF ME) (HIGHEST-IN-SUIT-LED CARDS-PLAYED1))

    (HAVE-POINTS CARDS-PLAYED1) =>
    (HAVE-POINTS (CARDS-PLAYED))
```

The first implication guarantees that when searching the space of potential scenarios (card sequences) for the trick to see if any cause player ME to take points, it is safe to ignore any sequences containing both player ME's card and a higher card in the suit led. The second implication suggests that a scenario in which player ME takes points might be found more quickly by giving priority to extending partial sequences that already have points, since their extensions will also have points. Deriving these implications involves a substantial amount of analysis.

In short, applying an operationalization method requires *domain knowledge* and *analysis* in addition to *knowledge about the method.*

1.7. Using the Heuristic Search Method

AI researchers draw upon a repertoire of sophisticated operationalization methods, but relatively little has been accomplished towards getting computers to apply them automatically. Mechanizing this process raises several issues:

How to represent a general method so that it can be reasoned about mechanically

How to map a particular problem onto the general problem statement for the method

How to fill in information required by the method but not explicit in the original problem

How to use additional knowledge about the problem domain to improve the solution

The dissertation explores these questions for the heuristic search method ("HSM") (§3). It presents some concrete answers in the form of a schema representation of HSM and some rules that operate on it. In DERIV2, these rules are used to operationalize the advice "avoid taking points." As a test of generality, the same schema and rules are used in DERIV13 to operationalize a simple music composition task.

1.7.1. A Generic Heuristic Search Procedure

Viewed abstractly, the purpose of heuristic search is to

Find a sequence of choices satisfying a given criterion.

The problem space for heuristic search is a set of partial sequences (*paths*). The search proceeds by selecting a path from the space, extending it by one of the permissible choices, and testing it to see if it satisfies the search criterion. This cycle repeats until a solution is found or the problem space is exhausted.

The components of FOO's HSM schema include the set of alternatives at each choice point and the search criterion expressed as a function of the choice sequence. Instantiating these components suffices to formulate an executable (albeit inefficient) search. For "avoid taking points," a path is a card sequence for the trick; the choice set at each point consists of the legal cards the player at that point might have, according to the information available to player ME; and the search criterion tests whether a given card sequence causes player ME to take points, *i.e.*, whether player ME's card is the highest in the suit led and the sequence contains one or more point cards.

1.7.2. Refining the Search

This inefficent search can be refined by reorganizing it so that constraints are applied as early as possible. In DERIV2, for example, the search is refined to generate only sequences in which player ME wins the trick, and to look first for sequences containing point cards, so as to quickly find a scenario in which player ME takes points. The schematic representation of HSM corresponds to a data flow graph containing various components that order or reject paths and alternatives. The search is refined by rules that transfer constraints from one such component to another.

1.7.3. Generality of the HSM Representation

The generality of the HSM schema (and the rules for instantiating and refining it) was tested on the music composition task described earlier (§1.2.4). The HSM rules developed to handle the Hearts example were either directly applicable to the music example or very similar to ones that were, and moreover led to some of the same design decisions embodied in a previous music composition program [Meehan 72].

1.8. Overview

The next several chapters describe knowledge used in operationalization. (§2) presents a repertoire of *operationalization strategies* and illustrates their use in evaluation and planning. (§3) shows how two problems from different domains are operationalized as *heuristic search* procedures, using many of the same general rules about the heuristic search method. (§4) discusses the process -- central to operationalization -- of *reformulating* a problem in terms of the capabilities of the task agent or the requirements of a method. (§5) describes the methods used for *analysis*; these provide the inferential power required to apply general operationalization methods to specific problems. (§6) examines the various ways in which general and domain-specific *conceptual knowledge* comes into play.

Next, (§7) sketches an approach for automatic problem-solving in the operationalization problem space based on means-end analysis, and presents a simulated example of its operation. (§8) discusses areas for further research. (§9) summarizes the contributions of the dissertation in light of previous research. Appendix A describes the hearts and music tasks. Appendix B lists FOO's transformation rules in a machine-generated notation and describes FOO's rule interpreter. Appendix C lists the concept properties and definitions in FOO's knowledge base. Appendix D presents the complete derivations, and Appendix E analyzes the operationality of the derived expressions.

Chapter 2
Operationalization Methods

This chapter presents several methods used in FOO to operationalize expressions. These include:

1. The pigeon-hole principle. (§2.2)
2. A probabilistic formula for estimating whether two subsets are disjoint. (§2.3)
3. Some rules for expressing present conditions in terms of past events. (§2.4)
4. A method for depleting a set. (§2.5)
5. A technique for finding necessary or sufficient conditions for a given predicate. (§2.6)
6. A model of choice. (§2.7)
7. A calculus for approximating an expression as a function of a known quantity. (§2.8)
8. The use of simplifying assumptions in planning and evaluation. (§2.9)
9. A strategy for integrating interdependent constraints. (§2.10)
10. A representation shift for treating a problem as a goal to be achieved over time. (§2.11)
11. A way to simplify a problem by binding one of its parameters before solving it. (§2.12)

These methods form the kernels of the derivations; they appear simple to the point of triviality, but are the key to solving a variety of problems, *given an analysis engine adequate to reformulate the problem in terms of the method and then solve the subproblems generated (explicitly or implicitly) by the method.* As Newell has observed,

> In general, in all design there is a key idea, which need not be very deep, but which changes an open problem of discovery to a closed task of problem solving.[5]

Each section of this chapter presents one of these methods, shows how it can be applied mechanistically to one or more problems, and describes its implementation as a transformation rule or collection of rules.

[5]Newell, personal communication.

The methods listed above represent a small, arbitrarily chosen set of points in the space of operationalization methods one could describe. Each method is formulated in a general fashion, *i.e.*, without reference to terms specific to card games or music. This *generality of expression* gives some hope for *generality of application*, *i.e.*, suggests that the methods might in fact apply not only to the problems used to illustrate them, but to other problems in other task domains. In a few cases, a method is explicitly applied to more than one problem.

2.1. Taxonomy of Operationalization Methods

FOO's operationalization methods can be classified according to some general criteria:

1. Purpose -- generate a procedure to *evaluate* an expression or *achieve* a goal
2. Scope -- how often can the procedure be executed?
3. Accuracy -- how often does executing the procedure produce the desired result?

There can be a tradeoff between scope and accuracy for a given method. The scope of a procedure can be defined as the proportion of situations in which it can be executed. The accuracy of a procedure can be defined as the correlation between its result and the correct answer, summed over all the situations in which it can be executed. This definition allows procedures that return results expressed as probabilities, *e.g.*, "player X is void with probability Y" or "play card X to avoid taking points with probability Y." The accuracy of such a procedure can be increased, at the cost of decreasing its scope, by only using the result in situations where the probability exceeds some threshold.

The criteria of purpose, scope, and accuracy define a taxonomy into which FOO's methods fit:

1. Methods for evaluating an expression
    a. Procedures that produce an exact result
        i. Procedures that always produce a result (assuming their inputs are available)

            Pigeon-hole principle (§2.2)

            Historical reasoning (§2.4)

            Heuristic search (§3)

        ii. Procedures that sometimes produce a result

            Check a necessary or sufficient condition (§2.6)

            Make a simplifying assumption that restricts the scope of applicability (§2.9.2.1)

iii. Procedures that produce an approximate result

Apply formula for probability of disjoint subsets (§2.3)

Functional dependence analysis (§2.8)

Use an untested simplifying assumption (§2.9.3)

Predict others' choices pessimistically (§2.7.3)

2. Methods for achieving a goal

a. Sound methods (introduce no errors) -- execution of plan (when feasible) will achieve goal

Set depletion (§2.5)

Find a sufficient condition and achieve it (§2.6)

Restrict a choice to satisfy the goal (§2.7.1)

Combine multiple goals (§2.10)

Rules for achieving goals over time (§2.11)

b. Heuristic methods -- execution of plan may not always achieve goal

Parameterize goal and bind parameter *a priori* (wrong value may make goal impossible) (§2.12)

Find a necessary condition and achieve it (§2.6)

Order choice set with respect to goal (§2.8.3)

2.2. The Pigeon-hole Principle

The pigeon-hole principle states that

A thing is in a given place if it isn't in any of the other places it could be.

A more general version of this principle is represented in FOO:

The value of a function lies in a given set if the rest of its range has been ruled out.

2.2.1. Example: Deciding if the Queen is Out

The pigeon-hole principle is used in DERIV4 to devise a way for deciding if the Queen of spades is out (*i.e.*, held by an opponent) without having to look at the opponents' cards:

```
(EVAL (OUT QS))
```

```
4:2-6 --- [by reformulation] --->

(EVAL (IN (LOC QS) (PROJECT HAND (OPPONENTS ME))))

4:7   --- [by RULE169] --->

(EVAL (NOT (IN (LOC QS)
               (DIFF (RANGE LOC)
                     (PROJECT HAND (OPPONENTS ME))))))

4:8-55     --- [by analysis] --->

(EVAL (NOT (OR [IN-POT QS] [AT QS HOLE] [HAS-ME QS] [TAKEN QS])))
```

None of these steps depends on any property specific to the Queen of spades; thus replacing every instance of the symbol QS in the derivation with, say, a variable CARD1, would produce a solution of the more general problem (EVAL (OUT CARD1)). An ideal operationalizer would make this observation automatically (§8.1.7). A similar technique was used in STRIPS to generalize a plan constructed to solve a specific problem by replacing problem constants with variables wherever possible [Fikes 71].

The fact that the same sequence of transformation rules would work for the more general problem justifies using the same solution in other cases requiring the ability to decide whether a card is out, without repeating the derivation for the more general case. For example, DERIV2 derives a necessary condition for whether an opponent P1 could possibly play card C2:

```
(HAS P1 C2) ; (=> (OUT C2) IF (IN P1 (OPPONENTS ME)))
```

This condition can be tested given a way to determine whether card C2 is out. In DERIV10, case analysis based on the pigeon-hole principle is used to derive an evaluable expression for the number of cards out in a given suit; this number is used to estimate the probability that an opponent is void in the suit.

The pigeon-hole principle is expressed in

RULE169: (in (f ...) S) -> (not (in (f ...) (diff (range f) S)))

2.2.2. Another Example: Prevent an Opponent from Getting the Lead

The pigeon-hole principle is not limited to finding ways to compute the OUT predicate. For instance, it could be used to account for the step

```
        (NOT (= (LEADER) (QSO)))
3:91    --- [by RULE353] --->
        (= (LEADER) ME)
```

This step transforms the problem of preventing the player who has the Queen of spades from getting the lead into the problem of getting the lead oneself. In DERIV3, this transformation was accounted for by the somewhat *ad hoc* rule

RULE353: (not (= e p)) -> (= e ME) assuming p ≠ ME

To prevent an opponent from taking on some role (i.e., unsharable position), do so yourself.

The same transformation could be accounted for instead by the sequence

```
(NOT (= (LEADER) (QSO)))

--- [by reformulation] --->
(NOT (IN (LEADER) (SET (QSO))))

--- [by RULE169, eliminating double negation] --->
(IN (LEADER) (DIFF (RANGE (LEADER)) (SET (QSO))))

--- [by analysis] --->
(IN (LEADER) (OPPONENTS (QSO)))

--- [by finding known element of set] --->
(= (LEADER) ME)
```

Here, RULE169 captures a game property analogous to the physical law

Two objects can't occupy the same space at the same time.

Specifically, two players can't be the leader of the same trick.

2.2.3. Pigeon-Hole Principle: Summary

Applying the pigeon-hole principle to a problem involves the following steps:

1. *Reformulate* the problem in terms of set membership.
2. *Transform* (in (f e) S) into (not (in (f e) S')) by RULE169, where S' is the complement of S with respect to the range of f.
3. *Simplify* the resulting expression to make it evaluable.

RULE169 itself does very little of the work; it just expresses the general idea. This is typical of rules that express ideas about how to operationalize. A possible problem-solving strategy (§8.1.1) for an operationalizer would use such rules as islands (subgoal generators) in the operationalization search space:

Try to map an operationalization problem onto known operationalization methods.

Such a strategy would need to be refined so as to consider only those methods appropriate for a particular problem. For example, there is no point in applying the pigeon-hole principle to an expression that is already evaluable.

2.3. The Disjoint Subsets Method

There is a general formula for the probability that two randomly-chosen subsets S, S' of a set U are disjoint:

(Pr-disjoint S S' U) = (#Choose |U-S'| |S|) / (#Choose |U| |S|)
where (#Choose n k) = n! / k!(n-k)!

This formula provides the basis for the following operationalization strategy:

Reformulate a predicate as an assertion that two subsets of a common superset are disjoint, and evaluate this assertion probabilistically by assuming that one of the subsets is chosen at random, independently of the other.

This strategy may be useful if the subsets are unevaluable but their sizes are known.

2.3.1. Example: Decide If an Opponent is Void

The disjoint-subsets method is used in DERIV9 to estimate the probability that an opponent `P0` is void in suit `S0`. Its application follows the scenario outlined in the Introduction (§1.6):

1. *Reformulate* the problem. (§2.3.1.1)
2. *Recognize* the method's applicability. (§2.3.1.2)
3. *Map* the problem onto the method. (§2.3.1.3)
4. *Fill in* additional information. (§2.3.1.4)
5. *Refine* the solution. (§2.3.1.5)
6. *Apply* the result. (§2.3.1.6)

2.3.1.1 Reformulate the Problem

The first step of the derivation reformulates the problem as a predicate on an unknown set:

```
        (VOID P0 S0)
9:1     --- [ELABORATE by RULE124] --->
        (NOT (EXISTS C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0)))
```

The set (CARD-IN-HAND P0) is generally unknown to player ME, since the rules of Hearts prohibit direct observation of opponents' hands.

2.3.1.2 Recognize the Method's Applicability

The second step recognizes that the disjoint subsets method is relevant, and reformulates the expression in terms of disjoint sets:

```
            (EXISTS C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0))
9:2-20      --- [by RULE189, analysis] --->
            (NOT (DISJOINT (CARDS-IN-HAND P0) (CARDS-IN-SUIT S0)))
```

The idea that the method may apply to a predicate on an unknown set is expressed in

RULE189: (P ... S ...) -> (h (disjoint S S')) for suitable h and S' if S is an unknown set

Here P = EXISTS, S = (CARDS-IN-HAND P0), h = NOT, and S' = (CARDS-IN-SUIT S0). As implemented, RULE189 does not actually check whether S is unknown, since FOO lacks a way to predict whether an expression will be evaluable at runtime (§8.1.1).

2.3.1.3 Map the Problem to the Method

The problem can now be reformulated to match the problem statement for the method:

```
        (DISJOINT (CARDS-IN-HAND P0) (CARDS-IN-SUIT S0))
9:21    --- [by RULE198] --->
        (PR-DISJOINT
           (CARDS-IN-HAND P0)
           (CARDS-IN-SUIT S0)
           (COMMON-SUPERSET (CARDS-IN-HAND P0) (CARDS-IN-SUIT S0)))
```

This step is accounted for by

RULE198: (disjoint S S') -> (Pr-disjoint S S' (common-superset S S')) if S is unknown, assuming S is randomly chosen independent of S' or *vice versa*.

No attempt was made to verify the randomness assumption by examining the rules of Hearts. In fact, this assumption is false if play has begun. Nonetheless, as with any method based on a simplifying assumption, the disjoint subsets method can be useful if the assumption is a reasonable approximation of reality. The more accurate the assumption, the better the result produced by the method.

2.3.1.4 Fill in Additional Information

The original problem didn't explicitly mention a common superset, so one must be found:

```
          (COMMON-SUPERSET (CARDS-IN-HAND P0) (CARDS-IN-SUIT S0)))
9:23-25   --- [by intersection search] --->
          (CARDS)
```

2.3.1.5 Refine the Solution

The probability that player P0 is void in suit S0 can be estimated by evaluating

```
(PR-DISJOINT (CARDS-IN-HAND P0) (CARDS-IN-SUIT S0) (CARDS))
= (#Choose 39 (#CARDS-IN-HAND P0)) / (#Choose 52 (#CARDS-IN-HAND P0))
```

However, this estimate takes too little information into account to be of much use: it varies according to the number of cards left in P0's hand but fails to discriminate between different suits. This solution can be improved by finding some property shared by (CARDS-IN-HAND P0) and considering only those cards that have it. This idea is expressed more precisely in

RULE200: (Pr-disjoint S S' U) -> (Pr-disjoint S (filter S' C) (filter U C)),
where C is a unary predicate satisfied by all of S, *i.e.*, such that S = (filter S C)

RULE200 searches the knowledge base for a suitable predicate C and finds the predicate OUT, originally used in the advice "if the Queen is out, flush it":

```
(PR-DISJOINT (CARDS-IN-HAND P0) (CARDS-IN-SUIT S0) (CARDS))
9:27-39 --- [REFINE by RULE200 (C = OUT), analysis] --->
(PR-DISJOINT (CARDS-IN-HAND P0) (CARDS-OUT-IN-SUIT S0) (CARDS-OUT))
```

RULE200's condition reduces to a restricting assumption:

```
(= (CARDS-IN-HAND P0) (FILTER (CARDS-IN-HAND P0) OUT))
9:28-36 --- [by analysis] --->
(IN P0 (OPPONENTS ME))
```

That is, all members of (CARDS-IN-HAND P0) are OUT *provided* P0 *is an opponent of* ME -- a restriction not made explicitly in the original problem statement (EVAL (VOID P0 S0)) (§5.9.3.2.2). The domain knowledge used in this analysis includes such information as the definition of S = (CARDS-IN-HAND P0), although the only information about S required by the probability formula is the size of S. This illustrates how the information required to *refine* a method may go beyond the information required to *apply* it.

2.3.1.6 Apply the Method

The method is *applied* by plugging in the formula:

```
(PR-DISJOINT
   (CARDS-IN-HAND P0)
   (CARDS-OUT-IN-SUIT S0)
   (CARDS-OUT))

9:40-43 --- [apply by RULE124, analysis] --->

(PR-DISJOINT-FORMULA
   (#CARDS-IN-HAND P0)
   (#CARDS-OUT-IN-SUIT S0)
   (#CARDS-OUT))
```

This expression estimates the probability that player P0 is void in suit S0, given the number of cards held by player P0, the number of cards out in suit S0, and the total number of cards out. These quantities can be computed by remembering which cards have been played (§2.4). Unlike the initial solution, this refined solution takes into account the cards in suit S0 held by player ME or previously played. An alternative to evaluating the expression directly to produce a numerical probability estimate is to operate on it by other methods (§2.8).

2.3.2. Another Example: Avoid Taking Points

Since the disjoint subsets method is potentially applicable to predicates on unknown sets, it's interesting to apply it to the "avoid taking points" example in the case where player ME must follow suit:

```
(AVOID-TAKING-POINTS (TRICK))

--- [by analysis, assuming (IN-SUIT-LED (CARD-OF ME))] --->

(EXISTS X1 (CARDS-PLAYED-BY-OPPONENTS)
   (AND [IN-SUIT-LED X1] [LOWER (CARD-OF ME) X1]))

--- [map to method by RULE189, analysis] --->

(NOT (DISJOINT (CARDS-PLAYED-BY-OPPONENTS)
               (CARDS-IN-SUIT-LED-ABOVE (CARD-OF ME))))

--- [fill in argument by RULE198, analysis] --->
```

```
(- 1 (PR-DISJOINT
        (CARDS-PLAYED-BY-OPPONENTS)
        (CARDS-IN-SUIT-LED-ABOVE (CARD-OF ME))
        (CARDS)))

--- [REFINE by RULE200 (C = IN-SUIT-LED), analysis] --->

(- 1 (PR-DISJOINT
        (CARDS-IN-SUIT-LED-PLAYED-BY-OPPONENTS)
        (CARDS-IN-SUIT-LED-ABOVE (CARD-OF ME))
        (CARDS-IN-SUIT-LED)))

        (CARDS-IN-SUIT-LED-PLAYED-BY-OPPONENTS) -- all of which were legal

        --- [by analysis of LEGAL] --->

        (CARDS-PLAYED-BY-NON-VOID-OPPONENTS) -- all of which were out

--- [REFINE by RULE200 (C = OUT), analysis] --->

(- 1 (PR-DISJOINT
        (CARDS-PLAYED-BY-NON-VOID-OPPONENTS)
        (CARDS-OUT-IN-SUIT-LED-ABOVE (CARD-OF ME))
        (CARDS-OUT-IN-SUIT-LED)))

--- [apply by RULE124, analysis] --->

(- 1 (PR-DISJOINT-FORMULA
        (#NON-VOID-OPPONENTS)
        (#CARDS-OUT-IN-SUIT-LED-ABOVE (CARD-OF ME))
        (#CARDS-OUT-IN-SUIT-LED)))
```

This expression estimates the probability that one of player ME's opponents will play a higher card (in the suit led) than (CARD-OF ME), *assuming that they choose randomly from their legal cards*, or more precisely that the cards played in the suit led are chosen randomly from those that are out. This unrealistic model is a built-in assumption of the method. A more realistic approach is to estimate the probability that all of player ME's non-void opponents *can* choose cards lower than (CARD-OF ME), rather than the probability that they will *randomly happen to do so*. The revised problem consists of finding a way to evaluate the assertion

```
(FORALL P1 (NON-VOID-OPPONENTS ME)
   (CAN-UNDERPLAY P1 (CARD-OF ME)))
```

Applying the disjoint subsets method to the quantified term might produce something like this:

```
(CAN-UNDERPLAY P1 (CARD-OF ME))

--- [by analysis] --->

(EXISTS X1 (CARDS-IN-SUIT-LED-IN-HAND P1)
   (LOWER X1 (CARD-OF ME)))

--- [map to method by RULE189, analysis] --->
```

```
(NOT (DISJOINT (CARDS-IN-SUIT-LED-IN-HAND P1)
               (CARDS-IN-SUIT-LED-BELOW (CARD-OF ME))))

--- [fill in argument by RULE198, analysis] --->

(- 1 (PR-DISJOINT
        (CARDS-IN-SUIT-LED-IN-HAND P1)
        (CARDS-IN-SUIT-LED-BELOW (CARD-OF ME))
        (CARDS-IN-SUIT-LED)))

--- [REFINE by RULE200 (C = OUT), analysis] --->

(- 1 (PR-DISJOINT
        (CARDS-IN-SUIT-LED-IN-HAND P1)
        (CARDS-OUT-IN-SUIT-LED-BELOW (CARD-OF ME))
        (CARDS-OUT-IN-SUIT-LED)))
```

Here some difficulties arise. First, (#CARDS-IN-SUIT-LED-IN-HAND P1) is generally unknown. Second, the obvious way to estimate the probability that all of plyaer ME's non-void opponents can underplay (CARD-OF ME) is to assume

```
(Pr [FORALL P1 (NON-VOID-OPPONENTS ME)
       (CAN-UNDERPLAY P1 (CARD-OF ME))])

= (PRODUCT P1 (NON-VOID-OPPONENTS ME)
     (PR-CAN-UNDERPLAY P1 (CARD-OF ME)))
```

However, this estimate is imperfect because the probabilities that different opponents P1 can underplay (CARD-OF ME) are not independent. For instance, if only one card lower than (CARD-OF ME) is still out, two different opponents can't both play it. Third, (#NON-VOID-OPPONENTS ME) is not directly evaluable, although its expected value can be estimated as

```
(SUM P1 (OPPONENTS ME)
   (PR-NON-VOID-IN-SUIT-LED P1))
```

Actually, the relevant quantity is the probability that the cards lower than (CARD-OF ME) are distributed so that every non-void opponent has at least one. A necessary condition for such a distribution is

```
(>= (#CARDS-OUT-IN-SUIT-BELOW (CARD-OF ME)) (#NON-VOID-OPPONENTS))
```

That is, there must be enough of them to go around. However, it is not obvious how to derive this condition mechanically from the original problem (AVOID-TAKING-POINTS (TRICK)). This may be a limitation of the method rather than of the apparatus for applying it. A more natural combinatorial model for this problem than the disjoint subsets model is the number of ways to put n balls in k slots without leaving any of them empty.

2.3.3. Disjoint Subsets Method: Summary

The disjoint subsets method provides a way to estimate the probability that a set S satisfies an assertion (P ... S ...). To apply the method, it is necessary to reformulate (P ... S ...) as a function of whether S is disjoint from some other set S', where S and S' are assumed to be chosen independently at random from some common universe U. The estimate provided by the method depends only on the sizes of the sets S, S', and U, and is therefore useful in evaluating assertions about sets of known size and unknown membership. The estimate can be refined if all members of S are known to satisfy some condition C, and the sets (filter S' C) and (filter U C) have known size.

2.4. Historical and Causal Reasoning

Sometimes evaluating an expression involves remembering previous events or conditions. Certain operationalization problems can be solved by deciding ahead of time to remember such information. For instance, it's a good idea to keep track of whether the Queen of spades has been played, who's void in what suit, and how many cards are out in each suit. This section describes rules for deciding what to remember; the solutions they produce contain "historical references" [Balzer 79] -- expressions defined in terms of past events. Implementing such solutions efficiently with appropriate demons and data structures is a separate problem that has been addressed by others [Barstow 77] [Low 78].

Another type of reasoning used in operationalization involves finding an action with a specified effect, such as causing a given predicate to become true or false. This is important both for constructing plans to achieve a desired effect and for reasoning from observed events to their consequences and *vice versa*.

Historical and causal reasoning are closely related, and it is useful to discuss them together. The examples in this section show among other things how one type of reasoning depends on the other.

2.4.1. Deciding if the Queen is Out

Historical reasoning is illustrated in the previously described problem of deciding whether the Queen of spades is out (§2.2.1). Applying the pigeon-hole principle to that problem splits it into several subproblems, one of which is deciding whether the Queen is in an opponent's pile:

```
(IN (LOC QS) (PILES (PLAYERS)))
4:44-46 --- [by analysis] --->
(EXISTS P3 (PLAYERS) (AT QS (PILE P3)))
```

Common sense says that if the Queen is in player P3's pile and it wasn't there at the beginning of the round, it had to get there somehow:

```
(AT QS (PILE P3))
4:47 --- [by RULE356] --->
(OR [WAS-DURING (CURRENT ROUND-IN-PROGRESS)
                (CAUSE (AT QS (PILE P3)))]
    [BEFORE (CURRENT ROUND-IN-PROGRESS) (AT QS (PILE P3))])
```

This reasoning is expressed more generally by

RULE356: P -> (or [was-during (current A) (cause P)] [before (current A) P]),
where A is an event type

The fact that players' piles are empty at the beginning of the round could in principle be inferred from the description of the game:

```
(BEFORE (CURRENT ROUND-IN-PROGRESS) (AT QS (PILE P3)))
4:48 --- [EVAL by RULE357] --->
NIL
```

However, here this inference is simulated by a Hearts-specific "fact rule" (§1.3.3):

RULE357: (before (current round-in-progress) (at c (pile p))) -> nil
for any card c and player p

Causing something to be somewhere means moving it there:

```
(CAUSE (AT QS (PILE P3)))
4:51 --- [RECOGNIZE by RULE358] --->
(MOVE QS LOC1 (PILE P3))
```

This general fact is expressed by

RULE358: (cause (at obj loc)) -> (move obj loc' loc)
where loc' is the previous location of the object obj

The game description mentions only one action that moves a card to a pile:

```
(MOVE QS LOC1 (PILE P3))
4:52 --- [RECOGNIZE by RULE123] --->
(TAKE P3 QS)
```

The resulting expression can be rewritten in terms of a recognized concept:

```
(EXISTS P3 (PLAYERS)
   (WAS-DURING (CURRENT ROUND-IN-PROGRESS) (TAKE P3 QS)))
4:53-54 --- [RECOGNIZE by RULE123] --->
(TAKEN QS)
```

It would be straightforward to implement this expression as a boolean value initialized to false at the beginning of the round and set to true by a demon when the Queen was taken. (However, this is beyond the scope of FOO, which doesn't know about data structures (§8.1.5) and provides no way to specify the time at which a given expression should be evaluated (§8.1.1).) This solution will work provided player ME observes the Queen being taken; if the Queen was the hole card (taken by the winner of the first trick and seen only by him), player ME will think it's still out, just as a human player might.

2.4.2. Remembering that an Opponent is Void

A player who happens to conclude that an opponent is void (*e.g.*, by noticing when he or she breaks suit) can remember this inference and use it later, since anyone who becomes void stays void for the rest of the round (although not for the rest of the game!). The idea of deciding whether a player is void by remembering if he or she was void earlier in the round is derived in DERIV12:

```
(VOID P0 S0)
12:1-13 --- [by RULE234, analysis] --->
(WAS-DURING (CURRENT ROUND-IN-PROGRESS) (VOID P0 S0))
```

This solution is found by looking for an event during which (VOID P0 S0) can't become false:

RULE234: P -> (was-during (current A) P)
if P is irreversible during events of type A, *i.e.*, (not (during (A) (undo P)))

Most of the derivation consists of verifying the rule condition for A = ROUND-IN-PROGRESS:

```
(NOT (DURING (ROUND-IN-PROGRESS) (UNDO (VOID P0 S0))))
12:2-12 --- [by analysis] --->
T
```

The proof of this condition begins by reformulating it in terms of causing a change in location:

```
(UNDO (VOID P0 S0))
12:2-8 --- [by analysis] --->
(EXISTS C1 (CARDS)
   (AND [CAUSE (AT C1 (HAND P0))] [IN-SUIT C1 S0]))
```

This enables the application of the previously described RULE358:

```
(CAUSE (AT C1 (HAND P0)))
12:9 --- [RECOGNIZE by RULE358] --->
(MOVE C1 LOC1 (HAND P0))
```

Moving a card to a player's hand is recognized as a special case of MOVE:

```
(MOVE C1 LOC1 (HAND P0))
12:10 --- [RECOGNIZE by RULE123] --->
(GET-CARD P0 C1 LOC1)
```

An intersection search (§5.6) through the knowledge base establishes that no card is moved to any player's hand while a round is in progress (*i.e.*, after the cards have been dealt out and any passing has taken place):

```
(DURING (ROUND-IN-PROGRESS)
   (EXISTS C1 (CARDS)
      (AND [GET-CARD P0 C1 LOC1] [IN-SUIT C1 S0])))
12:11 --- [COMPUTE by RULE57] --->
NIL
```

This completes the proof.

2.4.3. Keeping Count of the Cards

The expression derived earlier for estimating the probability that an opponent is void in suit S0 depends on the number of cards in S0 still out, *i.e.*, held by player ME's opponents. Without looking at their hands, how can this number be computed? The solution derived in DERIV10 is the formula 13 - M - P, where M is the number of cards in S0 held by ME when the round began, and P is the number of cards in S0 played so far by opponents of ME. This is done as follows.

First the expression is reformulated to fit

```
RULE370: (# (set-of x S Px)) ->
(- (# (set-of x S (before (current A) Px)))
   (# (set-of x S (was-during (current A) (undo Px)))))
if P is unachievable during events of type A, i.e., (not (during A (cause P)))
```

This rule counts the elements of a set that satisfy some property:

```
(#CARDS-OUT-IN-SUIT S0)
10:1-3 --- [ELABORATE by RULE124] --->
(# (SET-OF X1 (CARDS-IN-SUIT S0) (OUT X1)))
```

RULE370 exploits the fact that the number of cards out in suit S0 now is the number that were out when the round started, less those that have been played since then:

```
(# (SET-OF X1 (CARDS-IN-SUIT S0) (OUT X1)))
10:4-12 --- [by RULE370] --->
(- (# (SET-OF X1 (CARDS-IN-SUIT S0)
         (BEFORE (CURRENT ROUND-IN-PROGRESS) (OUT X1))))
   (# (SET-OF X1 (CARDS-IN-SUIT S0)
         (WAS-DURING (CURRENT ROUND-IN-PROGRESS)
                     (UNDO (OUT X1))))))
```

The validity of this reasoning depends on there being no way during the round for a card to *become* out, *i.e.*, enter an opponent's hand. Much of the derivation consists of verifying this condition, which comes from taking A = ROUND-IN-PROGRESS in RULE370:

```
(NOT (DURING (ROUND-IN-PROGRESS) (CAUSE (OUT X1))))
10:5-11 --- [by analysis] --->
T
```

Verification of the condition begins by reformulating it in terms of a change in location:

```
(CAUSE (OUT X1))
10:5-7 --- [by analysis] --->
(EXISTS X1 (CARDS)
   (CAUSE (AT X1 (HAND P0))))
```

This change of location is recognized as a moving a card to player P0's hand:

```
(CAUSE (AT X1 (HAND P1)))
10:8 --- [RECOGNIZE by RULE358] --->
(MOVE X1 LOC1 (HAND P1))
10:9 --- [RECOGNIZE by RULE123] --->
(GET-CARD P0 X1 LOC1)
```

A knowledge base intersection search establishes that no card can be moved to any player's hand while a round is in progress:

```
(DURING (ROUND-IN-PROGRESS)
   (EXISTS P1 (OPPONENTS ME)
      (GET-CARD P1 X1 LOC1)))
10:10 --- [COMPUTE by RULE57] --->
NIL
```

This completes the verification of RULE370's condition. The next part of the derivation simplifies the first term of the formula

```
(- (# (SET-OF X1 (CARDS-IN-SUIT S0)
         (BEFORE (CURRENT ROUND-IN-PROGRESS) (OUT X1))))
   (# (SET-OF X1 (CARDS-IN-SUIT S0)
         (WAS-DURING (CURRENT ROUND-IN-PROGRESS)
                     (UNDO (OUT X1))))))
```

First the pigeon-hole principle is applied, followed by some case analysis:

```
                (BEFORE (CURRENT ROUND-IN-PROGRESS) (OUT X1))
        10:14-22 --- [by analysis] --->
                (NOT (BEFORE (CURRENT ROUND-IN-PROGRESS) (HAS ME X1)))
```

The number of cards one *doesn't* have in a suit is 13 minus the number one *does* have:

```
          (# (SET-OF X1 (CARDS-IN-SUIT S0)
                (NOT (BEFORE (CURRENT ROUND-IN-PROGRESS) (HAS ME X1))))
        10:23-24 --- [by analysis] --->
          (- 13 (# (SET-OF X1 (CARDS-IN-SUIT S0)
                      (BEFORE (CURRENT ROUND-IN-PROGRESS) (HAS ME X1)))))
```

The rest of the derivation reformulates the second part of the formula in terms of causing a change in location:

```
        (# (SET-OF X1 (CARDS-IN-SUIT S0)
              (WAS-DURING (CURRENT ROUND-IN-PROGRESS)
                 (UNDO (OUT X1))
        10:25-28 --- [by analysis] --->
        (# (SET-OF X1 (CARDS-IN-SUIT S0)
              (WAS-DURING (CURRENT ROUND-IN-PROGRESS)
                 (EXISTS P3 (OPPONENTS ME)
                    (UNDO (AT X1 (HAND P3)))))))
```

This expression is recognized in terms of card-moving by applying the inverse of RULE358:

RULE368: (undo (at obj loc)) -> (move obj loc loc')
where loc' is the new location of the object obj

```
                    (UNDO (AT X1 (HAND P3)))
        10:29      --- [RECOGNIZE by RULE368] --->
                    (MOVE X1 (HAND P3) LOC2)
        10:30      --- [RECOGNIZE by RULE123] --->
                    (PLAY P3 X1)
```

The final result corresponds to the formula 13 - M - P, or (- (- 13 M) P) in prefix notation:

```
        (- (- 13 (# (SET-OF X1 (CARDS-IN-SUIT S0)
                       (BEFORE (CURRENT ROUND-IN-PROGRESS) (HAS ME X1)))))
           (# (SET-OF X1 (CARDS-IN-SUIT S0)
                 (WAS-DURING (CURRENT ROUND-IN-PROGRESS)
                        (EXISTS P3 (OPPONENTS ME) (PLAY P3 X1))))))
```

2.4.4. Reasoning about Actions from First Principles

The above derivations use RULE358 and RULE368, which relate motion to location. Note that RULE368 is also used in DERIV8 to help construct a plan for getting void in suit S0 by finding an action to get rid of a card:

```
        (REMOVE-1-FROM (SET-OF C1 (CARDS-IN-HAND ME) (IN-SUIT C1 S0)))
```

```
8:4 --- [REDUCE by RULE7] --->
(UNDO (AND [IN C1 (CARDS-IN-HAND ME)] [IN-SUIT C1 S0]))

8:5-9 --- [by analysis] --->
(AND [UNDO (AT C1 (HAND ME))] [IN-SUIT C1 S0])

8:10 --- [RECOGNIZE by RULE368] --->
(AND [MOVE C1 (HAND ME) LOC1] [IN-SUIT C1 S0])

8:11-14 --- [RECOGNIZE by RULE123, analysis] --->
(PLAY-SUIT ME S0)
```

These rules serve as shortcuts; it's actually possible to derive the same result from the definition

```
MOVE = (LAMBDA (OBJ ORIG DEST)
          (AND [= (LOC OBJ) ORIG] [BECOME (LOC OBJ) DEST]))
```

This is illustrated in the following excerpt from DERIV3, where the problem is to find an action that undoes player (QSO)'s possession of card C3. The approach taken is to try to instantiate an action concept selected from the knowledge base so as to have the desired effect:

```
(UNDO (HAS (QSO) C3))
3:46   --- [by RULE254] --->
(SHOW (=> [PLAY P1 C4] [UNDO (HAS (QSO) C3)]))
```

Here the selected concept is PLAY, whose arguments P1 and C4 are to be solved for. This idea is suggested by

RULE254: (C P) -> (A v_1 ... v_n)
if (A v_1 ... v_n) => (C P) for some action type A and argument values v_1 ... v_n,
where C is a modal operator (*e.g.*, "cause" or "undo") denoting change over time

As implemented, RULE254 lists the action concepts in the current task domain and asks the user to select one to use as A; an alternative would be to use trial and error to find one that works, *i.e.*, satisfies the rule condition for some assignment of values to its arguments.

In the current example, the process of solving for the arguments of PLAY begins by expanding the definitions of HAS and AT:

```
(SHOW (=> [PLAY P1 C4]
          [UNDO (HAS (QSO) C3)]))
3:47-48          --- [ELABORATE by RULE124] --->
                 (= (LOC C3) (HAND (QSO)))
```

Applying the definition of UNDO enables the recognition of MOVE:

```
                (UNDO (= (LOC C3) (HAND (QSO))))
3:49            --- [ELABORATE by RULE344] --->
                (AND [= (LOC C3) (HAND (QSO))] [BECOME (LOC C3) LOC3])
3:50            --- [RECOGNIZE by RULE123] --->
                (MOVE C3 (HAND (QSO)) LOC3)
```

The action PLAY can also be expressed in terms of MOVE:

```
                (PLAY P1 C4)
3:51            --- [ELABORATE by RULE124] --->
                (MOVE C4 (HAND P1) POT)
```

The two descriptions of MOVE events are matched, and their corresponding components equated:

```
            (=> [MOVE C4 (HAND P1) POT] [MOVE C3 (HAND (QSO)) LOC3])
3:52        --- [DISTRIBUTE by RULE43] --->
            (AND [= C4 C3] [= (HAND P1) (HAND (QSO))] [= POT LOC3])
```

The makes it possible to solve for the arguments P1 and C4 in (PLAY P1 C4). Plugging in the solved values yields an action with the desired effect (UNDO (HAS (QSO) C3))):

```
3:53-61 --- [by analysis] --->
(PLAY (QSO) C3)
```

2.4.5. Historical and Causal Reasoning: Summary

Knowledge about actions and their consequences is encoded in a variety of rules in FOO. It is useful to try to impose some structure on this collection.

2.4.5.1 A Model of Historical Reasoning

The three examples of historical reasoning (§2.4.1) (§2.4.2) (§2.4.3) used three different rules. However, all three are consequences of the same general relationship between the truth of a proposition P now and at a previous time t:

P is true now iff

P became true more recently than it became false

or P was true at time t and has not become false since then.

The condition "P became true more recently than it became false" is rather messy to encode, since it involves nested quantification over time. It can be considerably simplified given certain restrictions on the possibility of P changing. Each of the three rules follows from such a restriction:

RULE234: If P can't become false once it's true, it's true now if it was true at any earlier time.

RULE356: If P can't become false once it's true, it's true now *iff* it was true at time t or became true anytime since then.

RULE370: If P can't become true once it's false, it's true now *iff* it was true at time t and hasn't become false since then. (RULE370 also incorporates some knowledge about counting that could be encoded separately.)

Thus while the rules differ, they are accounted for by the same simple model.

2.4.5.2 Finding Actions with Specified Effects

FOO's rules about causality appear quite general with respect to the rather simple examples to which they were applied. The two rules for relating motion and location (RULE358 and RULE368) were pleasingly symmetric and used more than once. *Intersection* search through the space of defined action concepts was adequate in several instances to quickly show that one type of event couldn't occur during another. This technique relied on two properties of the knowledge base:

The *closure property* guarantees that the known concepts include all types of actions that can (legally) occur in the task environment. For instance, since the description of the first part of a trick mentions only that each player plays a card, FOO can safely assume that no poltergeist is secretly moving cards around at the same time.

The *canonical definition property* guarantees that

1. All the card-moving actions are ultimately defined in terms of MOVE.
2. No two action concepts defined directly in terms of MOVE (GET-CARD, PLAY, TAKE) can describe the same event.

This guarantees the type incompatibility of any two concepts with no sub-event concepts in common above the level of MOVE (§5.6). For instance, since undoing a void requires getting a card, and GET-CARD is not a sub-event of ROUND-IN-PROGRESS, FOO can conclude that a player who becomes void stays void for the rest of the round.

2.4.5.3 Significance

This section has presented some elementary but general techniques for reasoning about task domains involving change over time. *Historical reasoning* analyzes the current value of an expression in terms of its value at an earlier time and events that have occurred since that time. This type of reasoning is important in task domains where key features of the environment cannot be observed directly but must be inferred from observed behavior. *Causal reasoning* analyzes a change in the value of an expression in terms of the actions that could cause it. This type of reasoning is important in planning, where the object is to find an executable action that produces a desired effect. As AI addresses increasingly complex task domains, the utility of general mechanical methods for reasoning about them will increase accordingly.

2.5. Depleting a Set

One of the operationalization methods in FOO is

To achieve a goal of the form "make set S empty," remove one element at a time from S until the goal is accomplished.

This simple strategy underlies two plans derived using FOO: to get void in a given suit, keep playing cards from that suit; to flush the Queen, keep leading spades. In both cases, most of the work consists of reformulating the goal in terms of emptying a set, and then finding an executable action that removes an element from it.

2.5.1. Getting Void

A plan for getting void in a given suit S0 is derived in DERIV8. The derivation begins by reformulating the goal in terms of emptying a set:

```
(ACHIEVE (VOID ME S0))
8:1-2 --- [by analysis] --->
(ACHIEVE (EMPTY (SET-OF C1 (CARDS-IN-HAND ME)
                     (IN-SUIT C1 S0))))
```

This enables application of the general strategy for depleting a set:

```
8:3 --- [REDUCE by RULE6] --->
(UNTIL (VOID ME S0)
       (ACHIEVE (REMOVE-1-FROM (SET-OF C1 (CARDS-IN-HAND ME)
                                  (IN-SUIT C1 S0)))))
```

The set depletion strategy is expressed by

RULE6: (achieve (empty S)) -> (until P (achieve (remove-1-from S))),
where P is the original goal

Now the real work begins: figuring out how to remove an element from the set. Removing an element from {x in S | Px} means either removing an element of S or undoing Px for some x in S:

```
(REMOVE-1-FROM (SET-OF C1 (CARDS-IN-HAND ME) (IN-SUIT C1 SO)))
8:4 --- [by RULE7] --->
(UNDO (AND [IN C1 (CARDS-IN-HAND ME)] [IN-SUIT C1 SO]))
```

This general idea is expressed by

RULE7: (remove-1-from (set-of x S Px)) -> (undo (and [in x S] [Px])) for some x in S

Undoing a conjunction means undoing one of its conjuncts when the others were all true:

```
8:5 --- [REDUCE by RULE35] --->
(AND [UNDO (IN C1 (CARDS-IN-HAND ME))] [IN-SUIT C1 SO])
```

This reduction is suggested by

RULE35: (C (and $P_1 \ldots P_n$)) -> (and [C P_i] $P_1 \ldots P_{i-1}$ $P_{i+1} \ldots P_n$) for some P_i,
where C is a modal operator (*e.g.*, "cause" or "undo") denoting change

This rule says nothing about which P_i to choose. One obvious criterion is not to choose a P_i known to be immutable. For example, (IN-SUIT C1 SO) is permanently true or false for a given card C1 and suit SO, since a card can't change its suit. (In contrast, consider chess, where a pawn can become another piece.) This property could be derived from the domain knowledge base using the techniques described earlier (§2.4), or acquired simply by interrogating a domain expert. As implemented, RULE35 asks the user to select P_i. In the current example, P_i was chosen to be (IN C1 (CARDS-IN-HAND ME)). Alternatively, this choice could be discovered by trial and error, since no action in the Hearts domain can undo (IN-SUIT C1 SO).

Next (IN C1 (CARDS-IN-HAND ME)) is elaborated as a conjunction, and one of its conjuncts is chosen to be undone, once again using RULE35:

```
(UNDO (IN C1 (CARDS-IN-HAND ME)))
8:6-7 --- [by analysis] --->
(UNDO (AND [IN C1 (CARDS)] [HAS ME C1]))
8:8 --- [REDUCE by RULE35] --->
(AND [IN C1 (CARDS)] [UNDO (HAS ME C1)])
```

Here the alternative would be to make C1 stop being a card. Actions with this effect are physically possible (*e.g.*, burning), but are not part of the domain description.

The action of undoing (HAS ME C1) is now recognized in terms of the known action PLAY:

```
     (UNDO (HAS ME C1))
8:9-11 --- [by RULE368, analysis] --->
     (PLAY ME C1)
```

The action description (PLAY ME C1), combined with the other conditions, constitutes a known specialization of PLAY:

```
(AND [PLAY ME C1] [IN C1 (CARDS)] [IN-SUIT C1 S0])
8:12-14 --- [by analysis] --->
(PLAY-SUIT ME S0)
```

The resulting plan is to keep playing cards from S0 until void in S0:

```
(UNTIL (VOID ME S0) (ACHIEVE (PLAY-SUIT ME S0)))
```

Note that this plan must be executed within the confines of the game. Player ME can only perform one step of the plan (playing a card in suit S0) at each turn, and only when S0 is a legal suit to play.

Actually, the encoded problem (ACHIEVE (VOID ME S0)) is a simplified version of the original natural language advice "get void," which leaves open the choice of which suit(s) to get void in. To understand this advice at a deeper level one must appreciate its relation to other goals (§8.2.2) and the factors that affect the difficulty of implementing it. For example, getting void in suit S0 allows player ME to sluff (get rid of potentially undesirable cards) when S0 is led thereafter. This benefit does not occur if no other player will lead S0 -- for example, if player ME's opponents are already void in S0. The number of tricks it takes to get void in S0 (a measure of the difficulty of getting void) is at least the number of cards player ME holds in S0, but is also affected by the chances that another player will lead S0 or that player ME can do so.

An interesting problem would be to estimate (DIFFICULTY (ACHIEVE (VOID ME S0))) as a function of S0. In particular, it should be possible to derive the fact that this function increases with (#MY-CARDS-IN-SUIT S0) and decreases with (#CARDS-OUT-IN-SUIT S0). Even without knowing the exact function, this *functional dependence* information (§2.8) could be used to order suits according to the relative difficulty of getting void in them.

2.5.2. Flushing the Queen

A subtler use of the set depletion strategy occurs in DERIV3, where the problem is to devise a plan for flushing the Queen of spades into the open, *i.e.*, forcing its owner (QSO) to play it. As usual, the derivation begins by reformulating the initial expression in terms of the method:

```
        (FLUSH QS)
3:1     --- [ELABORATE by RULE124] --->
        (MUST (OWNER-OF QS) (PLAY (OWNER-OF QS) QS))
```

```
3:2-7      --- [by analysis] --->
           (= (LEGALCARDS (QSO)) (SET QS)))

3:12       --- [ELABORATE by RULE340] --->
           (AND [SUBSET (LEGALCARDS (QSO)) (SET QS)]
                [SUBSET (SET QS) (LEGALCARDS (QSO))])
```

At this point, the goal of flushing the Queen has been split into two subgoals. The first subgoal is for (QSO) to have no legal cards other than QS, and the second is for QS to be in fact legal for (QSO) to play. The second subgoal can be achieved by making sure that the suit led is spades:

```
                (SUBSET (SET QS) (LEGALCARDS (QSO)))
3:14-25         --- [by analysis] --->
                (= S (SUIT-LED))
```

The first subgoal is reformulated in terms of making a set empty:

```
                (SUBSET (LEGALCARDS (QSO)) (SET QS))
3:37            --- [by RULE4] --->
                (EMPTY (DIFF (LEGALCARDS (QSO)) (SET QS)))
```

The rule for expressing the SUBSET relation in terms of the EMPTY predicate is:

RULE4: (subset X Y) -> (empty (diff X Y))

This enables application of the set depletion strategy:

```
(ACHIEVE (EMPTY (DIFF (LEGALCARDS (QSO)) (SET QS))))
3:38 --- [REDUCE by RULE6] --->
(UNTIL (PLAYED! QS)
       (ACHIEVE
          (REMOVE-1-FROM (DIFF (LEGALCARDS (QSO)) (SET QS)))))
```

This step is performed by the same rule used in the "get void" example:

RULE6: (achieve (empty S)) -> (until P (achieve (remove-1-from S))),
where P is the original goal

Here, P = (PLAYED! QS) is filled in by the user. This reflects the following bit of goal knowledge lacking in FOO: the goal of flushing the Queen is to eliminate the risk of taking it, and this goal becomes satisfied as soon as the Queen is played, *regardless of whether the play is forced.* Filling in P by hand simulates the effect of retrieving a stored explanation of the reason for flushing the Queen. Such an explanation might be provided by the expert giving the advice, or, more interestingly, inferred from the task description (§8.2.3).

Implementing the set depletion strategy requires finding an action that removes an element from the set:

```
                (REMOVE-1-FROM (DIFF (LEGALCARDS (QSO)) (SET QS)))))
    3:39-87     --- [by analysis] --->
                (PLAY-SPADE (QSO))
```

Since this action has an agent other than player ME, a way is found to force it to happen:

```
          (ACHIEVE (PLAY-SPADE (QSO)))
    3:88-97 --- [by analysis of legal play] --->
          (ACHIEVE (LEAD-SPADE ME))
```

The final plan is to keep leading spades until the Queen is played:

```
    (UNTIL (PLAYED! QS) (ACHIEVE (LEAD-SPADE ME)))
```

Like the plan for getting void, this plan must be executed within the structure of legal play. In order to execute a step of the plan, player ME must be leading and have spades. A more sophisticated treatment of the advice "flush the Queen" would deal with such problems as getting and keeping the lead, and deciding whether player ME has enough spades to execute the plan to completion. Effective performance on these problems involves the ability to predict opponents' behavior based on knowledge of their goals. For example, the players who don't have the Queen are likely to share the goal of flushing it and cooperate in leading spades. This improves the chances of executing the plan.

Note that using (PLAYED! QS) as the termination condition for the plan avoids a couple of problems associated with the naive alternative of terminating when the event (FLUSH QS) occurs. This alternative condition is both non-operational and too strong. It's non-operational because player ME can't generally tell that player (QSO)'s only legal card is QS until after the fact, say after (QSO) plays QS and then breaks suit next time spades are led. The condition (FLUSH QS) is too strong because the motivation for leading spades ends as soon as the Queen is played, even if (QSO) has more spades left. This example illustrates the general phenomenon of *goal subsumption* [Wilensky 78]: if the only purpose of a goal G_1 is to help achieve a supergoal G_2, then the (possibly serendipitous) attainment of G_2 obviates the need to achieve G_1. This kind of understanding requires a representation of the relationships between goals, which FOO lacks (§8.2.3).

A defect of the derived plan is that it allows player ME to lead the Ace or King of spades. This is in fact a rather effective way to get the player holding the Queen to play it, but it defeats the underlying purpose of flushing the Queen, namely to avoid taking it. The general principle illustrated here is that any plan for achieving a subgoal G_1 of a goal G_2 should avoid violating G_2. When the goal of avoiding taking the Queen is added as an explicit constraint on the goal of flushing the Queen, a modified plan is derived (§2.10).

2.5.3. Set Depletion: Summary

Using the set depletion strategy to construct a plan involves the following steps:

1. *Reformulate* the goal in terms of making a set empty.
2. *Transform* (achieve (empty S)) into (until G (achieve (remove-1-from S))) by RULE6, where G is the original goal.
3. *Reformulate* (remove-1-from S) as an executable action A.

The operationality of the resulting plan depends on the ability of the plan agent to continue executing the action A until G is satisfied and to detect when this occurs.

2.6. Finding Useful Necessary or Sufficient Conditions

A general strategy for evaluating assertions is

If you can't evaluate an assertion directly, find an evaluable necessary or sufficient condition.

Such a condition enables partial evaluation of the original assertion A = (P ...): if a sufficient condition on A is true, A must be true; conversely, if a necessary condition on A is false, A must be false. Such a condition, along with any restrictions on its applicability, can be attached to the predicate P as an annotation to be used by runtime evaluation mechanism.

Similarly, a general strategy for achieving a goal is

If you can't achieve a goal directly, find an achievable necessary or sufficient condition.

Here, "achieving a goal directly" means translating it into an executable action. If no such action corresponds exactly to achieving the goal, the strategy suggests trying to solve an appropriate stronger or weaker goal instead. Strengthening the goal in the right way may make it easier to analyze and construct a plan for; a possible disadvantage is that such a plan may be infeasible in some situations where the original goal could have been achieved. Weakening the goal may make it easier to achieve, but doing so will not always have the desired effect of achieving the original goal. Thus goal-weakening is only effective to the extent that the additional conditions required to satisfy the original goal are likely to be true.

As stated above in their general form, these strategies doesn't tell *how* to find an evaluable necessary or sufficient condition for a given assertion (P ...). FOO finds possible candidates for necessary or sufficient conditions by looking in its knowledge base for predicates R that mention P (directly or indirectly) in their definitions. Subsequent logical analysis determines how to instantiate R's arguments and identifies any restrictions that must be satisfied for the instantiated predicate to be a necessary (or sufficient) condition on (P ...).

This section illustrates the first strategy by means of three derivations generated using FOO. The first derivation finds a sufficient condition for an opponent to be void: the opponent is breaking suit (§2.6.2). This condition can be evaluated by direct observation of the opponent's behavior. The other two derivations find necessary conditions for an opponent to have a card: the card must be out and the opponent can't be void in the suit of the card (§2.6.3). The former condition can be evaluated by remembering which cards have been played (§2.4.1). The latter condition is not always evaluable but can be useful in those situations where it is.

2.6.1. Implicit Decisions to Find Necessary or Sufficient Conditions

There are many other instances in the derivations where an expression is transformed into a necessary or sufficient condition. For instance, in DERIV3, the condition `(LEGAL (QSO) QS)` is reduced to the sufficient condition `(= (SUIT-LED) S)` by assuming that player `(QSO)` is following. The condition would also be satisfied if `(QSO)` were leading, but then `(QSO)`'s range of legal cards would be less constrained and hence harder for player `ME` to control. This example illustrates the usefulness of sufficient conditons not just in evaluating expressions but in constructing plans. However, in this and other examples the idea of replacing a predicate with a necessary or sufficient condition is *implicit* in the particular transformation rule, and limited to a *specific* condition rather than a collection of likely candidates found by searching the knowledge base of domain concepts. In contrast, the rules used in the following examples (RULE227 and RULE319) express a *general strategy* of deciding to look for a necessary or sufficient condition.

2.6.2. A Sufficient Condition for Deducing that an Opponent is Void

In DERIV11, the problem of deciding whether a player `PO` is void in suit `SO` is attacked using the strategy of looking for a sufficient condition, expressed by

RULE227: (eval (P ...)) -> (eval (=> [R x_1 ... x_n] [P ...])) if (R x_1 ... x_n) is true,
where R is some predicate with arguments x_1 ... x_n whose definition mentions P

FOO finds a predicate that mentions VOID in its definition, namely LEGAL =

```
(LAMBDA (P C)
   (AND [HAS P C]
        [=> (LEADING P)
            (OR [CAN-LEAD-HEARTS P] [NEQ (SUIT-OF C) H])]
        [=> (FOLLOWING P)
            (OR [VOID P (SUIT-LED)] [IN-SUIT C (SUIT-LED)])]))
```

RULE227 generates the subproblem of instantiating the arguments P1 and C1 so as to satisfy (LEGAL P1 C1):

```
          (SHOW (LEGAL P1 C1))
11:2      --- [FACT by RULE228] --->
          (SHOW (= C1 (CARD-OF P1)))
11:3-4    --- [by analysis] --->
(C1 <- BINDING : (CARD-OF P1))
          (SHOW T)
```

The condition (LEGAL P1 C1) is known to be true for C1 = (CARD-OF P1), *i.e.*, a card C1 played by a player P1 must be legal for P1 to play. This fact is encoded in a Hearts-specific "fact rule" (§1.3.3) to simulate the effect of analyzing the game description:

RULE228: (legal p c) -> T if (= c (card-of p))

Such an analysis might proceed by finding that LEGAL is mentioned (indirectly) in the definition of the action that determines (CARD-OF P), namely PLAY-CARD =

```
(LAMBDA (P)
   (CHOOSE (CARD-OF P) (LEGALCARDS P) (PLAY P (CARD-OF P))))
```

Thus (CARD-OF P) must satisfy the condition (IN (CARD-OF P) (LEGALCARDS P)), which reduces to (LEGAL P (CARD-OF P)). Hence (= C (CARD-OF P)) implies (LEGAL P C).

At this point, the condition (LEGAL P1 (CARD-OF P1)) has been formulated. It remains to determine when this condition implies (VOID P0 S0):

```
(EVAL (VOID P0 S0))
11:1-6 --- [by RULE227, analysis] --->
(EVAL (=> (LEGAL P1 (CARD-OF P1)) (VOID P0 S0)))
```

The rest of DERIV11 consists of reducing this implication to an evaluable predicate:

```
(EVAL (=> (LEGAL P1 (CARD-OF P1)) (VOID P0 S0)))

11:7-17 --- [by analysis] --->
(P1 <- BINDING : P0)
```

```
          (AND [NOT (IN-SUIT (CARD-OF P0) (SUIT-LED))]
               [= (SUIT-LED) S0]
               [FOLLOWING P0])
     11:18     --- [RECOGNIZE by RULE123] --->
     (EVAL (AND [BREAKING-SUIT P0] [= (SUIT-LED) S0] [FOLLOWING P0]))
```

This solution says that a sufficient condition for player P0 to be void in suit S0 is for P0 to be breaking suit when the suit led is S0 and (necessarily) P0 is following rather than leading. Thus upon seeing P0 break suit, player ME can conclude that P0 is void in the suit led -- a useful inference to remember, since it remains valid for the rest of the round (§2.4).

2.6.3. Two Necessary Conditions for a Player to Have a Card

A problem that arises in DERIV2 is deciding whether it's possible that a player P1 may have a card C2 (§3.3.4), where

An assertion is considered *possible* if none of its necessary conditions is known to be false.

Thus the more necessary conditions checked, the more discriminating the decision about possibility. A rule for finding necessary conditions is

RULE319: (P ...) <- ; (=> (Q x_1 ... x_n) IF R),
for some predicate Q(x_1 ... x_n) whose definition mentions P (perhaps indirectly),
where R is the result of simplifying (=> [P ...] [Q x_1 ... x_n])

(Recall that the notation "P_1 <- ; (=> P_2 IF R)" means "attach to the expression P_1 the annotation that P_2 is a necessary condition for P_1 when R is true.")

In the current example, (P ...) is (HAS P1 C2). One predicate whose definition mentions HAS is

```
     OUT = (LAMBDA (C) (EXISTS P (OPPONENTS ME) (HAS P C)))
```

RULE319 accordingly suggests (OUT C7) as a candidate necessary condition for (HAS P1 C2), where C7 is a new variable (to avoid name conflicts). It remains to solve for C7 and to determine when (HAS P1 C2) implies (OUT C7):

```
               (=> [HAS P1 C2] [OUT C7])
     2:29-38   --- [by analysis] --->
     (C7 <- BINDING : C2)
               (IN P1 (OPPONENTS ME))
```

Thus (OUT C2) is a necessary condition for (HAS P1 C2) if P1 is an opponent of player ME:

```
2:28-39    --- [by RULE319, analysis] --->
((HAS P1 C2) <- ;
       (=> (OUT C2) IF (IN P1 (OPPONENTS ME))))
```

Another predicate whose definition mentions HAS, albeit indirectly, is NON-VOID:

```
NON-VOID = (LAMBDA (P) (NOT (VOID P (SUIT-LED))))

VOID = (LAMBDA (P S)
         (NOT (EXISTS C (CARDS-IN-HAND P) (IN-SUIT C S))))

CARDS-IN-HAND = (LAMBDA (P) (SET-OF C (CARDS) (HAS P C)))
```

This fact is exploited by RULE319 in finding a second necessary condition for the assertion (HAS P1 C2). As before, applying RULE319 requires determining how to instantiate NON-VOID and figuring out when the instantiated predicate is a necessary condition on the original assertion:

```
          (=> [HAS P1 C2] [NON-VOID P5])
2:41-56    --- [by analysis] --->
(P5 <- BINDING : P1)
          (IN-SUIT C2 (SUIT-LED))
```

Thus (NON-VOID P1) is a necessary condition for (HAS P1 C2) if C2 is in (SUIT-LED):

```
2:40-57 --- [by RULE319, analysis] --->
((HAS P1 C2) <- ;
       (=> (NON-VOID P1) IF (IN-SUIT C2 (SUIT-LED))))
```

2.6.4. Finding Necessary or Sufficient Conditions: Summary

While RULE227 and RULE319 differ in form, they express symmetric ideas. If (P ...) is an assertion and Q(x_1 ... x_n) is a predicate whose definition mentions P directly or indirectly, then

(Q x_1 ... x_n) provides a *sufficient* condition for (P ...) when R is true, where R is the result of solving (=> [Q x_1 ... x_n] [P ...]) for x_1 ... x_n and simplifying.

(Q x_1 ... x_n) provides a *necessary* condition for (P ...) when R' is true, where R' is the result of solving (=> [P ...] [Q x_1 ... x_n]) for x_1 ... x_n and simplifying.

RULE227 solves for x_1 ... x_n so as to satisfy (Q x_1 ... x_n) and then reduces (P ...) to the sufficient condition R. RULE319 solves x_1 ... x_n to satisfy (=> [P ...] [Q x_1 ... x_n]) and then attaches to (P ...) the annotation that (Q x_1 ... x_n) is a necessary condition for (P ...) when R' is true. Such annotations can be useful in the absence of a systematic way to compute P.

2.6.4.1 Predictors of Rule Applicability

There is an interesting connection between the process of simplifying R and the restriction that Q's definition should mention P. The key step in simplifying R consists of eliminating the unevaluable predicate P in (=> [Q x_1 ... x_n] [P ...]) or (=> [P ...] [Q x_1 ... x_n]). In all three examples, this is accomplished by expanding Q in terms of P, and restructuring the expression into the form (... (=> [P e_1 ... e_n] [P e_1' ... e_n']) ...) so a partial-matching (unification) rule (§5.3) can be applied (recall that => is reflexive). The fact that a given predicate Q mentions P *in its definition* is a clue (although not a guarantee) that this simplification scenario may be feasible.

If FOO had to find Q by trial and error instead of asking the user to select it from a list of predicates mentioning P, such features -- cheaply-computable predictors of success -- would be extremely valuable. How might they be discovered automatically? One approach, inspired by Mitchell [Mitchell 81] and Hayes-Roth [Hayes-Roth 81b], would generalize from successful applications of a rule to scenarios describing their common structure, such as the following scenario for finding a necessary condition using RULE319:

1. *Choose* a predicate Q.
2. *Construct* the implication R.
3. *Expand* R in terms of P.
4. *Eliminate* P from R.
5. *Simplify* R.

In this approach, *unsuccessful* attempts to use a rule would be examined to find out where in the scenario they failed -- *e.g.*, couldn't expand Q in terms of P. The original rule could then be specialized to test for the property whose absence led to failure -- *e.g.*, test if Q is expandable to P. This test is potentially expensive, since all sorts of logical transformations may be performed in the course of expanding Q. A cheap substitute is to test whether Q mentions P in its definition. The new rule would constitute improved knowledge about how to operationalize, since it would lead to success more often than the unconstrained rule. Hayes-Roth has suggested a similar approach to refining knowledge *about a task domain* based on analysis of violated expectations [Hayes-Roth 81b]; the novel aspect here is the idea of refining knowledge about operationalization in general. Easier said than done, of course.

2.7. Reasoning About Choice

FOO uses a simple representation of choice based on the quantifier `CHOOSE`. For example, the expression `(CHOOSE (CARD-OF P) (LEGALCARDS P) (PLAY P (CARD-OF P)))` represents the action "player `P` plays a card, `(CARD-OF P)`, chosen from `P`'s legal cards." Similarly, the expression `(CHOOSE (NOTE I) (TONES) (NOTE I))` represents "a note, `(NOTE I)`, chosen from the set `(TONES)`." In general, the construct (choose x S E_x) means "the entity E_x, where x is chosen from the set S." Thus the construct has the same type as the quantified sub-expression E_x; in particular, it may denote an event or an object. The quantifier variable x may be parameterized to distinguish among similar choices made at different points. For example, `(CARD-OF P)` distinguishes player `P`'s card from the cards chosen by other players. Similarly, `(NOTE I)` distinguishes the Ith note of a *cantus firmus* from the other notes in the sequence.

The quantifier `CHOOSE` introduces an element of non-determinism into the semantics of FOO's representation. The modal operators "necessary" and "possible" can be used to form deterministic predicates from non-deterministic choice-dependent predicates:

An assertion that depends on a choice among alternatives is *possible* if true for at least one alternative and *necessary* if true for all alternatives.

In particular,

(possible (P (choose x S E_x))) = (exists x S (P E_x))

(necessary (P (choose x S E_x))) = (forall x S (P E_x))

Some of FOO's rules about choice essentially transform a choice-dependent assertion A to (possible A) or (necessary A) depending on who makes the choice. In the rules shown below, the contruct (choose-me x S E_x) denotes a choice made by the agent executing a plan generated using FOO, while (choose-other x S E_x) denotes a choice made by another agent. However, this distinction is only implicit in FOO's actual representation.

1. *Constrained choice:* (achieve (P (choose-me x S E_x))) -> (choose-me x [set-of x S (P E_x)] E_x)
   To achieve a goal that depends on something you choose, choose it so as to satisfy the goal.
2. *Forcing:* (achieve (P (choose-other x S E_x))) -> (achieve (necessary (P (choose-other x S E_x))))
   To achieve a goal that depends on someone else's choice, make all his alternatives satisfy it.
3. *Pessimism:* (eval (P (choose-other x S E_x))) -> (eval (possible (P (choose-other x S E_x))))
   Evaluate an undesirable condition as the possibility that an opponent can make it true.

The rest of this section illustrates the actual or potential use of each of these rules.

2.7.1. Constraining a Choice to Satisfy a Goal

Some of the derivations operationalize goals as evaluable constraints on a choice variable. For example, DERIV6 operationalizes "avoid taking points" as

```
(ACHIEVE (=> (AND [IN-SUIT-LED (CARD-OF ME)]
                  [TRICK-HAS-POINTS])
             (LOW (CARD-OF ME))))
```

This expression means "if you're following suit and the trick has points, make your card low." (Assume that the predicate `TRICK-HAS-POINTS` is operationalized elsewhere, *e.g.*, as a probabilistic estimate derived using the disjoint subsets method (§2.3)). Here the value of the choice variable `(CARD-OF ME)` is determined by the choice event

```
(PLAY-CARD ME) =
   (CHOOSE (CARD-OF ME) (LEGALCARDS ME) (PLAY ME (CARD-OF ME)))
```

Similarly, DERIV7 operationalizes "get the lead" as

```
(ACHIEVE (AND [IN-SUIT-LED (CARD-OF ME)] [HIGH (CARD-OF ME)]))
```

This plan can be paraphrased as "make your card a high card in the suit led."

Since these expressions are (fuzzy) predicates on a choice variable, they could be integrated with other similarly operationalized goals by means of an evaluation function. Each legal choice for `(CARD-OF ME)` could be scored according to how well it satisfied each fuzzy predicate. The scores for each goal could be added together, perhaps weighted by the relative importance of the goals in the current game situation [Berliner 79]. (Knowledge about relative goal importance would have to be acquired somehow, *e.g.*, from an expert.) The card with the best total score would be chosen.

An alternative to such an integration mechanism is to transform constraints on choice variables into executable action descriptions, using the "constrained choice" transformation. To achieve the goal, one simply makes sure to choose an element of the choice set that satisfies the constraint:

RULE156: (achieve P_x) -> (choose-me x [set-of y S P_y] E_x)
where x is determined by the event (choose-me x S E_x)

For example, this transformation can be applied to the constraint

```
(ACHIEVE (=> (AND [IN-SUIT-LED (CARD-OF ME)]
                  [TRICK-HAS-POINTS])
             (LOW (CARD-OF ME))))
```

Here x = `(CARD-OF ME)` is determined by the event `(PLAY-CARD ME)` =

```
(CHOOSE (CARD-OF ME) (LEGALCARDS ME) (PLAY ME (CARD-OF ME)))
```

The executable action produced by the constrained choice transformation is:

```
(CHOOSE (CARD-OF ME)
        (SET-OF Y (LEGALCARDS ME)
           (=> (AND [IN-SUIT-LED Y] [TRICK-HAS-POINTS]) (LOW Y)))
        (PLAY ME (CARD-OF ME)))
```

This action description can be paraphrased as "choose a legal card -- make sure it's a low card if it's in the suit led and the trick has points -- and play it."

Note that the naming scheme for choice variables gives a unique mapping from a choice variable to the event that determines it. For example, the parameterized name `(CARD-OF P)` denotes the card chosen during the event `(PLAY-CARD P)`, and is not used as a choice variable in any other events.. This distinguishes between cards played by different players. Since the example derivations only dealt with one trick at a time, it was unnecessary to distinguish between cards played in different tricks, but this could be done by using an extra parameter `E` to specify the trick: `(CARD-OF P E)`.

The prerequisite to applying the constrained choice transformation is to

Reformulate the original goal as an evaluable predicate on a choice variable.

This is what constitutes all the work in DERIV6 and DERIV7.

2.7.2. Restricting Another's Options to Satisfy a Goal

DERIV3 derives a plan to flush the Queen by eliminating its owner's other legal choices. Knowledge about controlling others' choices is expressed by the "forcing" transformation

(achieve (P (choose-other x S E_x))) -> (achieve (necessary (P (choose-other x S E_x))))

If P has the form (lambda (x) (= x e)), the right side can be simplified:

(necessary (P (choose-other x S E_x)))
-> (forall x S (P E_x)) -- *by definition of necessary*
-> (forall x S (= E_x e)) -- *by definition of P*
-> (= S {e}) -- *assuming S is non-empty*

This special case of the forcing transformation is given by

RULE339: (= [choose x S (act agent x)] [act agent obj]) -> (= S {obj})

RULE339 generates the goal of reducing player (QSO)'s legal cards to the singleton set {QS}:

```
                (ACHIEVE (FLUSH QS))
3:1-5                     --- [by analysis] --->
                (ACHIEVE (= [PLAY-CARD (QSO)] [PLAY (QSO) QS]))
3:6                          --- [ELABORATE by RULE124] --->
                (ACHIEVE (= [CHOOSE (CARD-OF (QSO))
                                    (LEGALCARDS (QSO))
                                    (PLAY (QSO) (CARD-OF (QSO)))]
                            [PLAY (QSO) QS]))
3:7                       --- [by RULE339] --->
                (ACHIEVE (= (LEGALCARDS (QSO)) (SET QS)))
3:12-87         --- [by set depletion, analysis] --->
                (UNTIL (PLAYED! QS) (ACHIEVE (PLAY-SPADE (QSO))))
```

Applying RULE339 followed by the set depletion method (§2.5) reduces the problem to the subproblem of repeatedly forcing player (QSO) to play a spade. One might expect this subproblem to be solved by applying a more general form of the forcing transformation, *e.g.*:

RULE339a: P_x -> (forall x S P_x), where x is determined by (choose-other x S E_x)

Such a solution might look like this:

```
(ACHIEVE (PLAY-SPADE (QSO)))
--- [by finding a sufficient condition] --->
(ACHIEVE (=> [PLAY-CARD (QSO)] [PLAY-SPADE (QSO)]))
--- [by analysis] --->
(ACHIEVE (SPADE! (CARD-OF (QSO))))
--- [by RULE339a, analysis] --->
(ACHIEVE (FORALL X (LEGALCARDS (QSO)) (SPADE! X)))
--- [by analysis of LEGAL] --->
(ACHIEVE (AND [FOLLOWING (QSO)] [= (SUIT-LED) S]))
```

However, the rule actually used in DERIV3 to solve this subproblem was the somewhat kludgey

RULE351: (achieve P) -> (achieve (and A_1 ... A_n)),
where A_1 ... A_n are the assumptions made in reducing the original goal to P

This rule enabled communication between two conjunctive subgoals (§5.4): achieving a state in which player `(QSO)` could legally play `QS`, and *eliminating* `(QSO)`*'s other spades*. The conditions assumed in solving the first subgoal, *i.e.*, that `(QSO)` wouldn't be leading and that spades would be led, sufficed to satisfy the second subgoal (a fortuitous happenstance not explicitly proved in DERIV3):

```
(ACHIEVE (PLAY-SPADE (QSO)))
3:88-89 --- [by RULE351, simplification] --->
(ACHIEVE (AND [NOT (LEADING (QSO))] [= (SUIT-LED) S]))
```

RULE351 essentially compensated for an inadequacy of FOO's problem-solving apparatus.

2.7.3. Predicting an Opponent's Choice

The knowledge about choice described so far deals with *controlling* choices, *i.e.*, achieving goals that depend on choices made by oneself or someone else. The same simple model of choice is useful in *predicting* choices, *i.e.*, evaluating choice-dependent predicates. This is illustrated by the "pessimistic transformation":

(eval (P (choose-other x S E_x))) -> (eval (possible (P (choose-other x S E_x))))

To evaluate an undesirable condition that depends on someone else's choice, assume the worst.

This assumption is implicit in the minimax search method for game trees [Nilsson 71], and underlies many counterplanning strategies [Carbonell 79]. The pessimistic transformation can be expressed as a sort of dual of RULE339a, *e.g.*,

RULE339b: P_x -> (exists y S P_y),
where P_x is undesirable and x is determined by (choose-other x S E_x)

This transformation is not explicitly used in the derivations, but can be illustrated by a couple of hypothetical examples. The first of these, introduced earlier (§2.3), is the problem of predicting whether player `ME` will win the trick when following suit:

```
(FORALL C1 (CARDS-PLAYED-IN-SUIT) (NOT (HIGHER C1 (CARD-OF ME))))
--- [by analysis] --->
(FORALL P1 (OPPONENTS ME) (LOWER (CARD-OF P1) (CARD-OF ME)))
--- [by RULE339b] --->
(FORALL P1 (OPPONENTS ME)
   (EXISTS X1 (LEGALCARDS P1) (LOWER X1 (CARD-OF ME))))
```

This expression is still not operational, but is easier to evaluate in the sense that it does not depend on choices made by player ME's opponents, only on what cards they have. It is consequently susceptible to evaluation by other methods, *e.g.*, the disjoint subsets method.

A similar example is the problem of predicting whether the trick will have points, assuming player ME plays a card without points:

```
(TRICK-HAS-POINTS)
--- [ELABORATE by RULE124] --->
(EXISTS C1 (CARDS-PLAYED) (HAS-POINTS C1))
--- [by analysis] --->
(EXISTS P1 (OPPONENTS ME) (HAS-POINTS (CARD-OF P1)))
--- [by RULE339b] --->
(EXISTS P1 (OPPONENTS ME)
   (EXISTS Y1 (LEGALCARDS P1) (HAS-POINTS Y1)))
```

Here the undesirable event mentioned in RULE339b is `(HAS-POINTS (CARD-OF P1))`, where `(CARD-OF P1)` is determined by the event `(PLAY-CARD P1) =`

```
(CHOOSE (CARD-OF P1) (LEGALCARDS P1) (PLAY P1 (CARD-OF P1)))
```

The effect of applying RULE339b is to assume that the trick will have points if any of player ME's opponents can legally play a point card. This pessimistic assumption is rather naive, but a more realistic treatment would require knowledge about players' goals and behavior. As in the previous example, the expression produced by RULE339b is non-operational but choice-independent and susceptible to further operationalization. For instance, the disjoint subsets method might be used to evaluate the sub-expression `(EXISTS Y1 (LEGALCARDS P1) (HAS-POINTS Y1))` as a probability `(PR-LEGAL-POINT-CARD P1)`. If this probability is assumed independent for different opponents P1, the overall expression could be evaluated as

```
(- 1 (PRODUCT P1 (OPPONENTS ME) [- 1 (PR-LEGAL-POINT-CARD P1)]))
```

2.7.4. Model of Choice: Summary

FOO uses the quantified construct (choose x S E_x) to represent an entity E_x involving a variable x chosen from a set S of alternatives. Quantifier variable names are parameterized so as to distinguish among choices made at different points. The construct does not explicitly identify the agent A making the choice, although it is an implicit argument in the following rules for achieving or evaluating conditions involving choice:

1. *Constrained choice:* (achieve (P (choose x S E_x))) -> (choose x [set-of x S (P E_x)] E_x) if A = ME
   To achieve a goal that depends on something you choose, choose it so as to satisfy the goal.
2. *Forcing:* (achieve (P (choose x S E_x))) -> (achieve (forall x S (P E_x))) if A ≠ ME
   To achieve a goal that depends on someone else's choice, make all his alternatives satisfy it.
3. *Pessimism:* (eval (P (choose x S E_x))) -> (eval (exists x S (P E_x))) if A ≠ ME
   Evaluate an undesirable condition as the possibility that an opponent can make it true.

2.8. Approximation based on Functional Dependence

An underlying problem in some of the derivations is to predict the probability of an event, *e.g.*, that a given card will win the trick, or to estimate the probability of a condition, *e.g.*, whether an opponent is void in a given suit. Such probabilities may be affected by many factors, and consequently hard to compute exactly. A way to simplify this problem is to choose one of these factors and analyze its effect. This approach underlies the method of *functional dependence analysis:*

Approximate an expression as a function of its dependence on a given quantity.

Although the examples in this section involve probability, the method itself is phrased in more general terms. It provides a 0^{th}-order approximation for an expression that is too expensive to compute exactly, or that can't be directly evaluated because some of its arguments are unavailable.

For example, the probability of winning the trick can be approximated as a function of the rank of the card one plays. This approximation might be represented as follows:

```
(= (TRICK-WINNER) ME)
--- [by approximation] --->
(FUNCTION-OF (RANK (CARD-OF ME)) INCREASING)
```

Here the construct (function-of v increasing) means "an increasing function of v". Similarly, (function-of v decreasing) means "a decreasing function of v".

2.8.1. A Calculus of Functional Dependence

The *functional dependence* of an expression e on a quantity v describes how e changes when v increases: increasing, decreasing, constant, or indeterminate. It is denoted by the construct (dependence e v) and can be thought of as sign($\partial e/\partial v$). For instance, the above example involves the computation

```
(DEPENDENCE [= (TRICK-WINNER) ME] [RANK (CARD-OF ME)])
--- [by functional dependence analysis] --->
INCREASING
```

The value of (dependence e v) is defined to be a member, d, of the set {0, ↑, ↓, ?}, where

d = NIL (0) if e stays constant as v increases

d = INCREASING (↑) if e increases as v increases

d = DECREASING (↓) if e decreases as v increases

d = INDETERMINATE (?) otherwise

The parenthesized symbols will be used for abbreviation, as will the following notation:

$(D_v\ e) = (\text{dependence } e\ v)$

$Df = (D_x\ (f\ x))$

$Df_i = Df_{x_i} = (D_{x_i}\ (f\ x_1 \ldots x_n))$ where $x_1 \ldots x_n$ are the arguments of f

However, FOO uses the long form, which appears in the derivations.

The four values {0, ↑, ↓, ?} form a commutative "pseudo-ring" with pseudo-inverse D-, addition D+, and multiplication D* defined as shown in Figure 2-1. A similar algebra was used by DeKleer [DeKleer 79] but lacked the D* operator.

This system has additive identity 0 and multiplicative identity ↑:

$(D{+}\ d\ 0) = d$
$(D^*\ d\ \uparrow) = d$

It is associative and commutative, so D+ and D* can be extended to take n arguments:

$(D{+}\ d) = d$
$(D{+}\ d_1 \ldots d_n) = (D{+}\ d_1\ (D{+}\ d_2 \ldots d_n))$
$(D^*\ d) = d$
$(D^*\ d_1 \ldots d_n) = (D^*\ d_1\ (D^*\ d_2 \ldots d_n))$

It fails to be a group under addition because (D+ d (D- d)) is not 0 for d = ↑, ↓, or ?.

The following identities are useful in evaluating $(D_v\ e)$ for a given expression e:

$(D_v\ (f\ e)) = (D{-}\ (D_v\ e))$ if $Df = \downarrow$, *e.g.*, $(D_v\ {-e}) = (D{-}\ (D_v\ e))$

$(D_v\ (f\ e)) = (D^*\ Df\ (D_v\ e))$, *e.g.*, $(D_x\ (f\ (g\ x))) = (D^*\ Df\ Dg)$ -- this follows from the chain rule $\partial(f\ (g\ x))/\partial x = (\partial f/\partial g)(\partial g/\partial x)$

D-	0	↑	↓	?
	0	↓	↑	?

D+	0	↑	↓	?
0	0	↑	↓	?
↑	↑	↑	?	?
↓	↓	?	↓	?
?	?	?	?	?

D*	0	↑	↓	?
0	0	0	0	0
↑	0	↑	↓	?
↓	0	↓	↑	?
?	0	?	?	?

Figure 2-1: Calculus of Functional Dependence

$(D_v\,(f\,e_1\,e_2)) = (D{+}\,(D_v\,e_1)\,(D_v\,e_2))$ if $Df_1 = Df_2 = \uparrow$,
e.g., $(D_v\,(+\,e_1\,e_2)) = (D{+}\,(D_v\,e_1)\,(D_v\,e_2))$

The last identity should be qualified as follows. If F is the space of functions from S_1 to S_2, where S_1 and S_2 are ordered sets, the operator D_v is almost a homomorphism from F to $\{0, \uparrow, \downarrow, ?\}$, but fails to satisfy the identity

$$(D_v\,(+\,e_1\,e_2)) = (D{+}\,(D_v\,e_1)\,(D_v\,e_2))$$

An example that violates this identity is

$$(D_v\,(+\,v\,{-v})) = (D_v\,0) = 0 \text{ but } (D{+}\,(D_v\,v)\,(D_v\,{-v})) = (D{+}\,\uparrow\,\downarrow) = ?$$

However, D_v does satisfy the weaker condition

$$(D_v\,(+\,e_1\,e_2)) = (D{+}\,(D_v\,e_1)\,(D_v\,e_2))\ \textit{provided}\,(D{+}\,(D_v\,e_1)\,(D_v\,e_2)) \neq 0$$

Most of FOO's dependence analysis rules are special cases of the generalized chain rule

$$(D_v\,(f\,e_1 \ldots e_n)) = (D{+}\,[D^*\,Df_1\,(D_v\,e_1)] \ldots [D^*\,Df_n\,(D_v\,e_n)])$$

What is all this apparatus good for? The rest of this section presents an explicit example of its use, followed by a couple of examples that turn out to involve the same underlying idea.

2.8.2. An Example of Functional Dependence Analysis

In DERIV9, the disjoint subsets method is used to estimate the probability that an opponent PO is void in suit SO as the quantity

```
(PR-DISJOINT-FORMULA
   (#CARDS-IN-HAND PO)
   (#CARDS-OUT-IN-SUIT SO)
   (#CARDS-OUT))
```

Suppose the purpose of estimating this quantity is to apply the advice

"Avoid leading a suit in which someone is void."

To choose the safest suit, one need not know the precise probability of a void in each suit; it is sufficient to know which suits are more likely than others to have voids. Functional dependence analysis can rank these probabilities without computing their numerical values. This would be very useful if the probability formula were expensive to compute. The decision to apply this technique appears in DERIV9 as

```
(EVAL (PR-DISJOINT-FORMULA (#CARDS-IN-HAND PO)
      (#CARDS-OUT-IN-SUIT SO)
      (#CARDS-OUT)))
9:44 --- [by RULE202] --->
(FUNCTION-OF SO (DEPENDENCE
                  (PR-DISJOINT-FORMULA (#CARDS-IN-HAND PO)
                                       (#CARDS-OUT-IN-SUIT SO)
                                       (#CARDS-OUT))
                  SO))
```

This decision is motivated by

RULE202: (eval e) -> (function-of v (D_v e)),
where v is some variable in the original problem

To implement this decision, it is necessary to analyze the functional dependence involved:

```
          (DEPENDENCE (PR-DISJOINT-FORMULA (#CARDS-IN-HAND PO)
                                           (#CARDS-OUT-IN-SUIT SO)
                                           (#CARDS-OUT))
                      SO)
9:45-46   --- [by RULE203] --->
          (D* (DEPENDENCE (#CARDS-OUT-IN-SUIT SO) SO)
              (DEPENDENCE (PR-DISJOINT-FORMULA #H #S #U) #S))
```

The analysis begins by using the variant chain rule

RULE203: $(D_v (f\, e_1 \ldots e_n)) \text{->} (D+ \ldots [D^* Df_i (D_v\, e_i)] \ldots)$,
where the sum (D+ ...) includes a term for each e_i in which v occurs (since the others go to nil)

Here f = PR-DISJOINT-FORMULA

v = SO

$\{x_i\}$ = {#H, #S, #U}

$\{x_i$ | v occurs in $e_i\}$ = {#S}

This is followed by the simplification (D+ d) -> d.

The alert reader may have noticed that the variable SO ranges over the unordered set of suits, rendering expressions like (DEPENDENCE (#CARDS-OUT-IN-SUIT SO) SO) rather meaningless. (What is the partial derivative of an expression with respect to a suit?) This is now fixed by transforming the approximation from a function of the nominal (non-numerical) variable SO to a function of the integer-valued expression (#CARDS-OUT-IN-SUIT SO):

```
(FUNCTION-OF SO
             (D* (DEPENDENCE (#CARDS-OUT-IN-SUIT SO) SO)
                 (DEPENDENCE (PR-DISJOINT-FORMULA #H #S #U) #S)))

9:47 --- [by RULE210] --->

(FUNCTION-OF (#CARDS-OUT-IN-SUIT SO)
             (D* INCREASING
                 (DEPENDENCE (PR-DISJOINT-FORMULA #H #S #U) #S)))

9:48         --- [by multiplicative identity] --->

(FUNCTION-OF (#CARDS-OUT-IN-SUIT SO)
             (DEPENDENCE (PR-DISJOINT-FORMULA #H #S #U) #S))
```

Further analysis is based on the definition

(Pr-disjoint-formula #H #S #U) =
(#choose [- #U #S] #H) / (#choose #U #H)

The fact that x/y is an increasing function of x and a decreasing function of y is retrieved from the generalization hierarchy (§1.3.2.1), which shows that division belongs to the category INCR1-DECR2. This justifies the application of

RULE205: $(D_v\ (f\ e_1\ e_2))$ -> $(D+\ (D_v\ e_1)\ (D\text{-}\ (D_v\ e_2)))$ if $Df_1 = \uparrow$ and $Df_2 = \downarrow$

RULE205 is a special case of the chain rule:

```
                   (DEPENDENCE (PR-DISJOINT-FORMULA #H #S #U) #S))
9:50               --- [by RULE205] --->
                   (D+ (DEPENDENCE (#CHOOSE (- #U #S) #H) #S)
                       (D- (DEPENDENCE (#CHOOSE #U #H) #S)))
```

The fact that (#CHOOSE #U #H) is independent of #S permits the application of

RULE204: $(D_v\ e)$ -> nil if v does not occur in e (assuming e doesn't expand to v)

This leads to some simplification:

```
                           (DEPENDENCE (#CHOOSE #U #H) #S)
9:51                       --- [EVAL by RULE204] --->
                           NIL

                       (D- NIL)
9:52                   --- [COMPUTE by RULE236] --->
                       NIL

                   (D+ (DEPENDENCE (#CHOOSE (- #U #S) #H) #S) NIL)
9:53               --- [SIMPLIFY by RULE343] --->
                   (DEPENDENCE (#CHOOSE (- #U #S) #H) #S)
```

The fact that $D\#choose_1 = \uparrow$ (*i.e.*, (#choose n k) increases with n) is encoded as an is-a link from #CHOOSE to INCR1. This fact justifies the reduction

```
                   (DEPENDENCE (#CHOOSE (- #U #S) #H) #S)
9:54               --- [REDUCE by RULE207] --->
                   (DEPENDENCE (- #U #S) #S)
```

This step uses another special case of the chain rule:

RULE207: $(D_v\ (f\ e_1 \ldots e_n))$ -> $(D_v\ e_1)$ if $Df_1 = \uparrow$ and v does not occur in $e_2 \ldots e_n$

Subtraction is-a INCR1-DECR2, so RULE205 is applied once again:

```
                   (DEPENDENCE (- #U #S) #S)
9:55               --- [by RULE205] --->
                   (D+ (DEPENDENCE #U #S) (D- (DEPENDENCE #S #S)))
9:56-59            --- [by RULE204, simplification] --->
                   (D- (DEPENDENCE #S #S))
```

The fact that any quantity is an increasing function of itself is encoded in

RULE208: $(D_v\ v) \rightarrow \uparrow$

This fact is used in completing the functional dependence analysis:

```
                  (D- (DEPENDENCE #S #S))
     9:57         --- [EVAL by RULE208, simplification] --->
                  DECREASING
```

2.8.3. Application: Ordering a Choice Set

The final approximation derived for `(VOID P0 S0)` is

```
    (FUNCTION-OF (#CARDS-OUT-IN-SUIT S0) DECREASING)
```

This expression could be used to order the suits according to the estimated riskiness of leading them. A simple way to do this is to define

(function-of v d) $= +v$ if $d = \uparrow$
(function-of v d) $= -v$ if $d = \downarrow$
(function-of v d) $= 0$ otherwise

A refinement normalizes the range to be [0,1]:

(function-of v d) $= (v - v_{min}) / (v_{max} - v_{min})$

Computing the normalized function would require determining v_{min} and v_{max}; a different normalization scheme would be required if either of these were infinite. In any case, the function of v thus constructed could be used as a term in an evaluation function used to *order the choice set* of legal cards (§2.7.1).

2.8.4. Analysis Combined with Experience

An interesting application of functional dependence analysis would be possible in a system with mechanisms not only for taking advice but for learning from its own experience. Instead of using a linear function of v as a 0th-order approximation of the original expression e, the task agent could

Compile a table of the actual values of e encountered for each value of v.

In the earlier example (§2.8.2), this would mean keeping track of how often an opponent turned out to be void when a given number of cards in the suit were still out. (Note that this approach requires a way to evaluate e, although it may be evaluated after the fact, *i.e.*, after the situation where the value of e was needed.) Given enough data, the system would eventually develop a good empirical estimate of the probability of a void as a function of the number of cards out in the suit. Human players appear to acquire knowledge of game probabilities by a similar process; it seems hard otherwise to explain how they discover heuristics like

> A suit is generally safe for the first two tricks in which it's led.

The approach advocated here is to

> *Use dependence analysis to identify potential predictors of important events and conditions, and determine their predictive strength empirically.*

This approach should help reduce the combinatorics inherent in "bottom-up" discovery of such predictors or criterial features of the task domain, while avoiding the often intractable task of computing their predictive strength analytically. *I.e.*, analysis may *guide* empirical discovery.

2.8.5. Extensions

DERIV9 is the only derivation involving the explicit use of functional dependence analysis. The same technique could be used to approximate other expressions (not just probabilities!) as functions of observable quantities. For example, the expected number of points in a trick could be estimated as a function of the number of points still out:

```
(#POINTS-IN-TRICK)
--- [by RULE202, functional dependence analysis] --->
(FUNCTION-OF (#POINTS-OUT) INCREASING)
```

An extension of the method would approximate an expression as a function of n quantities:

$e \rightarrow$ (function-of v_1 $(D_{v_1}\ e)$ ... v_n $(D_{v_n}\ e)$)

For example, a better estimate for the expected number of points in a trick might be given by

```
(FUNCTION-OF (#POINTS-OUT) INCREASING
             (#CARDS-OUT-IN-SUIT-LED) DECREASING)
```

The n-ary function might be computed as a linear combination of $v_1 \ldots v_n$, or as an n-dimensional tables containing empirically determined values of e for different values of $v_1 \ldots v_n$.

2.8.6. Generality

The rules used in DERIV9 included some *ad hoc* specializations of the generalized chain rule:

RULE205: $(D_v\ (f\ e_1\ e_2))$ -> $(D+\ (D_v\ e_1)\ (D-\ (D_v\ e_2)))$ if $Df_1 = \uparrow$ and $Df_2 = \downarrow$

RULE207: $(D_v\ (f\ e_1 \dots e_n))$ -> $(D_v\ e_1)$ if $Df_1 = \uparrow$ and v does not occur in $e_2 \dots e_n$

These specialized rules served to shorten the derivation by avoiding consideration of several terms that would have reduced to nil. A more efficient implementation might use a systematic procedure for dependence analysis, based on the generalized chain rule.

2.8.7. Implicit Dependence Analysis

A second look at the "avoid taking points" example reveals the underlying idea of functional dependence. In DERIV6, the goal `(AVOID-TAKING-POINTS (TRICK))` is reformulated as a condition containing the sub-expression

```
(EXISTS X1 (CARDS-PLAYED-IN-SUIT-LED) (LOWER (CARD-OF ME) X1))
```

Since the set `(CARDS-PLAYED-IN-SUIT-LED)` is generally unknown when `(CARD-OF ME)` is chosen, this sub-expression is approximated as the fuzzy condition `(LOW (CARD-OF ME))` by

RULE154: $(R\ e_1\ e_2)$ -> $(P\ e_1)$, where R is the comparative form of predicate P

An alternative version of this derivation would be

```
(AVOID-TAKING-POINTS (TRICK))

--- [by RULE202] --->

(FUNCTION-OF (CARD-OF ME)
             (DEPENDENCE (AVOID-TAKING-POINTS (TRICK))
                         (CARD-OF ME)))

--- [by RULE210, functional dependence analysis] --->

(FUNCTION-OF (LOW (CARD-OF ME)) INCREASING)
```

Thus RULE154 can be expressed in functional dependence notation as the approximation rule

$(R\ e_1\ e_2)$ -> (function-of $(P\ e_1)$ increasing)

RULE154 is also used in the course of constructing a plan to get the lead:

```
(ACHIEVE (LEADING ME))
7:1-7 --- [by analysis, RULE154] --->
(ACHIEVE (AND [IN-SUIT-LED (CARD-OF ME)]
              [NOT (LOW (CARD-OF ME))]))
```

Explicit functional dependence analysis might produce this essentially similar solution:

```
(ACHIEVE (LEADING ME))
--- [by RULE210, functional dependence analysis] --->
(FUNCTION-OF (RANK (CARD-OF ME)) INCREASING)
```

2.8.8. Functional Dependence Analysis: Summary

An alternative to evaluating an expression e is to analyze its *functional dependence* on some known quantity v, using a simple calculus of 4 elements $\{\uparrow, \downarrow, 0, ?\}$. Since this technique requires less information than direct evaluation, it may be applicable when some of e's arguments are unknown, or when e is too expensive to compute. It gives a way to order a set $\{e_v \mid v \text{ in } V\}$ in terms of an ordering on the set V, without actually evaluating e_v for each v.

2.9. Simplifying Assumptions

A problem can be made easier to solve by making suitable assumptions. Some strategies for making assumptions include:

Assume that an unlikely, undesirable, or complicating condition is false.

Assume that a desirable and plausible condition is true.

If (f e) is difficult to evaluate, but (f x) = c for most x in the domain of f, assume (f e) = c.

2.9.1. Implicit Simplifying Assumptions

The derivations use many implicit assumptions of the form "assume this transformation preserves semantic equivalence." For instance, consider the partial matching rule

RULE43: $(R\ (f\ e_1 \ldots e_n)\ (f\ e_1' \ldots e_n')) \rightarrow (\text{and } [= e_1\ e_1'] \ldots [= e_n\ e_n'])$ if R is reflexive

This rule preserves logical equivalence (is *valid*) *assuming certain conditions on R and f* (§5.3.3). In particular, the function f must be *injective, i.e.*, produce a different value for every distinct combination of values to which it's applied. Otherwise, RULE43 is only *sound, i.e.*, its right hand side implies its left hand side but the converse may not hold. The derivations often implicitly assume

RULE43, and various other rules, to be valid. This corresponds to assuming certain sometimes subtle properties of the task domain. For example, one step in reformulating the advice `(AVOID (TAKE-POINTS ME) (TRICK))` as `(ACHIEVE (LOW (CARD-OF ME)))` is:

```
          (DURING [TAKE (TRICK-WINNER) C3] [TAKE ME C1])
     6:11 --- [REDUCE by RULE43] --->
          (AND [= (TRICK-WINNER) ME] [= C3 C1])
```

To assume that this transformation preserves logical equivalence is to assume that only one TAKE event can occur at a time. The significance of implicit rule validity assumptions as a form of knowledge about the task domain is discussed further in (§6.4).

In contrast, this section discusses the use of *explicit* assumptions in evaluating expressions and in constructing plans to achieve goals. It gives examples of how to make, retrieve, and apply them.

2.9.2. *Assumptions* in Planning

A useful strategy in achieving a goal is to

Treat an assumption made in constructing a plan as an additional subgoal.

For instance, in DERIV3, a subgoal of flushing the Queen of spades is to make it legal for the player (QSO) who has the Queen to play it. The real problem, not represented explicitly, is to satisfy this subgoal *as narrowly as possible, i.e.*, make the Queen one of a small set of legal options. If player (QSO) is leading, he or she can play almost any card. Since the goal is to control (QSO)'s choice of card, the case in which (QSO) is following is more promising. This reasoning is based on a general strategy (used elsewhere in counter-planning [Carbonell 78]):

To control a choice made by someone else, narrow the range of alternatives.

It uses the specific Hearts knowledge that

A player's range of legal cards is usually narrower when following than when leading.

The latter empirical fact is rather hard to prove rigorously from the rules of the game, since it depends on the relative frequency of exceptions, such as when a player has lots of hearts but can't lead them because hearts haven't been broken yet. However, the same plan can still be discovered without knowing this fact. Consider the goal

```
                          (LEGAL (QSO) QS)
3:20-25                   --- [by analysis] --->
                          (AND [=> (LEADING (QSO))
                                   (OR [CAN-LEAD-HEARTS (QSO)]
                                       [NEQ (SUIT-OF QS) H])]
                               [=> (FOLLOWING (QSO))
                                   (OR [VOID (QSO) (SUIT-LED)]
                                       [= S (SUIT-LED)])])
```

The case where (QSO) is following involves the condition (= S (SUIT-LED)). Making (SUIT-LED) spades is a significant restriction, since (SUIT-LED) can take on any of four values:

```
(# (RANGE SUIT-LED)) = (# (SUITS)) = 4
```

The derivation therefore proceeds by assuming this restriction, as suggested by

RULE342: (= c e) -> assume e = c, where c is a constant

As implemented, RULE342 does not try to compute the range of f in (= c (f ...)). The decision to apply RULE342 here reflects my knowledge of Hearts. The same decision might be made mechanically by trial and error: in the alternative case, player (QSO) leads and need not play QS.

The assumption that spades are led suffices to achieve the subgoal (LEGAL (QSO) QS):

```
                          (=> (FOLLOWING (QSO))
                              (OR [VOID (QSO) (SUIT-LED)]
                                  [= S (SUIT-LED)]))
3:26-34                   --- [by RULE342, analysis] --->
                          T
ASSUMING (SUIT-LED) = S
```

Later in DERIV3, the set depletion method (§2.5) is applied to the subgoal of making the Queen of spades player (QSO)'s only legal card:

```
(SUBSET (LEGALCARDS (QSO)) (SET QS))

3:37-38 --- [by RULE6, analysis] ->

(UNTIL (PLAYED! QS)
       (ACHIEVE
          (REMOVE-1-FROM (DIFF (LEGALCARDS (QSO)) (SET QS)))))

3:39-61 --- [by analysis] --->
```

```
(UNTIL (PLAYED! QS)
       (ACHIEVE (AND [PLAY (QSO) C3]
                     [=> (LEADING (QSO))
                         (OR [CAN-LEAD-HEARTS (QSO)]
                             [NEQ (SUIT-OF C3) H])]
                     [=> (FOLLOWING (QSO))
                         (OR [VOID (QSO) (SUIT-LED)]
                             [IN-SUIT C3 (SUIT-LED)])]
                     [IN C3 (DIFF (CARDS) (SET QS))])))
```

This expression is simplified by substituting S for (SUIT-LED) based on the earlier assumption:

```
                     (=> (FOLLOWING (QSO))
                         (OR [VOID (QSO) (SUIT-LED)]
                             [IN-SUIT C3 (SUIT-LED)]))
3:62-79              --- [by RULE346, analysis] --->
                     (=> (FOLLOWING (QSO)) (IN-SUIT C3 S))
```

The rule for substitution based on an assumed equality is

RULE346: e -> c if e = c was assumed earlier

Steps 3:64-77 of the analysis eliminate the impossible clause (VOID (QSO) S).

The problem is complicated by the case (LEADING (QSO)), so this case is assumed to be false:

```
                     (=> (LEADING (QSO))
                         (OR [CAN-LEAD-HEARTS (QSO)]
                             [NEQ (SUIT-OF C3) H]))
3:80                 --- [ASSUME by RULE349] --->
                     T
ASSUME (LEADING (QSO)) = NIL
```

This assumption illustrates the general strategy

Assume that an unlikely, undesirable, or complicating condition is false.

In this instance the strategy is expressed by

RULE349: (=> P Q) -> T plus the assumption that P is false,
where P is a case you'd like to rule out

Here I played the role of problem-solver and identified the case to be ruled out; FOO has no way to do so itself. The assumption serves to simplify the remaining case:

```
                     (=> (FOLLOWING (QSO)) (IN-SUIT C3 S))
```

```
                              (FOLLOWING (QSO))
     3:83                     --- [ELABORATE by RULE124] --->
                              (NOT (LEADING (QSO)))
     3:84-85.                 --- [EVAL by RULE346, analysis] --->
                              T

                          (=> T (IN-SUIT C3 S))
     3:86                 --- [SIMPLIFY by RULE12] --->
                          (IN-SUIT C3 S)
```

Both assumptions are then adopted as subgoals:

```
          (ACHIEVE (PLAY-SPADE (QSO)))
     3:88 --- [by RULE351] --->
          (ACHIEVE (AND [= (LEADING (QSO)) NIL] [= (SUIT-LED) S]))
```

This step is suggested by the slightly kludgey rule

RULE351: (achieve P) -> (achieve (and P_1 ... P_n)),
where P_1 ... P_n are the assumptions made in reducing the original goal to P

The kludge is the assumption that P_1 ... P_n are sufficient to guarantee P, *i.e.*, that RULE351 is sound. If not, it may lead to a plan that doesn't always achieve the original goal (§2.6). Here, however, satisfying the two subgoals is adequate to make player (QSO) play a spade. The eventual plan is:

```
     (UNTIL (PLAYED! QS) (PLAY-SPADE ME))
```

As mentioned elsewhere, this plan is only partly operational, since in order to execute it player ME must have the lead and enough spades to last until player (QSO) is forced to play QS. The point of the example, however, is to illustrate how assumptions are *made*, *retrieved*, and *applied* in constructing the plan. The assumptions that spades are led and that (QSO) is following are *made* on the basis of general rules for making simplifying assumptions. Assumptions of the form <expression> = <constant> are *retrieved* when the <expression> shows up elsewhere in the problem, and are *applied* to simplify the problem by substituting the <constant> for the <expression>. Finally, the assumptions are adopted as subgoals, whose solution gives a plan for achieving the original goal.

2.9.2.1 Applications of Planning Assumptions

Assumptions made in planning can be used in several ways. One represented by RULE351 is to

Incorporate the assumption as a subgoal.

For instance, the ability of player ME to determine the suit led generally requires that player ME be the leader. This condition can added to the plan for flushing the Queen as a subgoal:

Get the lead and play a spade.

An alternative to satisfying the assumptions as part of a plan is to

Incorporate the assumption as a restriction on the applicability of the plan.

Thus having the lead can be taken as a precondition of the plan for flushing the Queen:

If you're leading, play a spade.

A third possibility is useful when a plan involves an uncertain contingency. A simple way to deal with this is to

Assume the contingency will occur and plan accordingly.

This idea is discussed in (§2.9.2.2).

2.9.2.2 Contingency Planning

The contingency planning approach is illustrated in DERIV5, where the plan for flushing the Queen is modified to avoid taking it in the process. This is done by assuming the Queen will be played and planning accordingly:

```
(AND [ACHIEVE (FLUSH QS)] [AVOID (TAKE ME QS) (TRICK)])
5:1-35 --- [by RULE301, analysis] --->
(UNTIL (PLAYED! QS)
       (ACHIEVE (AND [LEAD-SPADE ME]
                     [=> [IN QS (CARDS-PLAYED)]
                         [EXISTS X1 (CARDS-PLAYED-IN-SUIT-LED)
                            (HIGHER X1 (CARD-OF ME))]]))))
```

This expression can be paraphrased as

Lead spades until the Queen is played -- but when the Queen is played, make sure your card isn't the winning card (highest card played in the suit led).

Playing a card lower than the Queen will suffice to keep player ME from winning the trick if the Queen is played:

```
                   (=> [IN QS (CARDS-PLAYED)]
                       [EXISTS X1 (CARDS-PLAYED-IN-SUIT-LED)
                          (HIGHER X1 (CARD-OF ME))])
5:39-49            --- [by RULE84, analysis] --->
                   (=> [IN QS (CARDS-PLAYED)]
                       [HIGHER QS (CARD-OF ME)])
```

Player ME can't generally predict when the Queen will be played. Contingency planning is reflected in the decision to assume it will be played in the current trick:

```
                         (=> [IN QS (CARDS-PLAYED)]
                             [HIGHER QS (CARD-OF ME)])
5:50                     --- [by RULE157] --->
                         (HIGHER QS (CARD-OF ME))
```

This assumption is suggested by the converse of RULE349:

RULE157: (=> P Q) -> Q, where Q is a goal and P is an uncertain contingency

The simplification based on this assumption leads to the plan

```
(UNTIL (PLAYED! QS) (ACHIEVE (LEAD-SAFE-SPADE ME)))
```

2.9.3. Assumptions in Evaluation

So far, the discussion of simplifying assumptions has focussed on their use in planning. A suitable assumption can also be used to simplify an evaluation problem, at the cost of producing an inaccurate value when the assumption is wrong:

Approximate an expression by making a suitable assumption.

If the assumption is one whose truth can be relative, *e.g.*, "the cards are randomly distributed," the accuracy of the value may depend on the degree to which the assumption is satisfied. The tradeoff of accuracy for simplicity can be worthwhile, especially if the original expression is hard to analyze. The trick, of course, is finding a "suitable assumption."

An example of such an assumption is the decision in DERIV10 to ignore the hole card when computing the number of cards out in suit S0. Before this decision, this number has been shown to be the number that were out when the round started, minus those no longer out:

```
(EVAL (#CARDS-OUT-IN-SUIT S0))
10:1-12 --- [by RULE370, analysis] --->
(EVAL (- (# (SET-OF X1 (CARDS-IN-SUIT S0)
               (BEFORE (CURRENT ROUND-IN-PROGRESS) (OUT X1))))
         (# (SET-OF X1 (CARDS-IN-SUIT S0)
               (WAS-DURING (CURRENT ROUND-IN-PROGRESS)
                           (UNDO (OUT X1)))))))
```

To compute the number of cards out at the start of the round, the pigeon-hole principle is used, plus the fact that the deck and pot are empty when the round begins:

```
                (BEFORE (CURRENT ROUND-IN-PROGRESS) (OUT X1))))
10:14-18                --- [by RULE301, analysis] --->
                (OR [BEFORE (CURRENT ROUND-IN-PROGRESS)
                            (AT X1 HOLE)]
                    [BEFORE (CURRENT ROUND-IN-PROGRESS)
                            (HAS ME X1)])
```

At this point, the problem is simplified by assuming the hole card wasn't in the suit led:

```
                            (AT X1 HOLE)
10:19                       --- [ASSUME by RULE373] --->
                            NIL
10:20-21        --- [by analysis] --->
                (BEFORE (CURRENT ROUND-IN-PROGRESS)
                        (HAS ME X1))
```

This assumption is generated by the Hearts-specific rule

RULE373: (at card hole) -> nil by assumption

It is based on the knowledge that any given card is unlikely to be the hole card. (Note that the plausibility of this assumption is compromised by quantification over `X1`; however, it's not entirely invalidated if the scope of quantification covers a reasonably small fraction of the set of cards. In this example, the quantifier covers a fourth of the cards.) A more general rule would be

RULE373a: P -> nil by assumption *if* P is unlikely

However, this would require the ability to estimate (Pr P), or at least to verify (unlikely P).

An even more general version of RULE373a is the strategy

RULE373b: *If (f e) is difficult to evaluate, but (f x) = c for most x in the domain of f, assume (f e) = c.*

In this example, f = `(LAMBDA (C) (AT C HOLE))`, which is `NIL` for all but one card.

2.9.4. Simplifying Assumptions: Summary

Simplifying assumptions can be very useful in both planning and evaluation. A suitable assumption may help reduce an expression to a simpler one, or even to a constant, typically by replacing a sub-expression with its assumed value. The assumption can then be treated as a subgoal, as a plan contingency, or as a restriction on the applicability of the solution. Alternatively, it can be kept as an untested condition whose violation may occasionally invalidate the solution.

2.10. Combining Interdependent Constraints

This section discusses strategies for operationalizing conjunctions of goals. As such, it touches on the issue of *integration* (§8.2.2), which has otherwise been excluded as beyond the scope of the dissertation.

2.10.1. Using an Assumption Made in Planning

A common problem in constructing plans to achieve multiple goals arises when the goals interact. For example, consider the two goals generated in DERIV3 in the course of constructing a plan to flush the Queen of spades:

```
(ACHIEVE (AND [SUBSET (LEGALCARDS (QSO)) (SET QS)]
              [SUBSET (SET QS) (LEGALCARDS (QSO))]))
```

The second goal is to make the Queen a legal card for the player who has it. This goal is easily satisfied if that player has the lead. However, that makes it difficult to satisfy the first goal of eliminating the player's other legal cards.

One approach to this problem is

Record the assumptions made in solving one goal and apply them to the others.

Assumptions in FOO have the form <expression> = <value> and are applied by substituting <value> for instances of <expression> occurring in the other goals (§5.4.3).

The example is solved by reducing the second goal to the assumption that spades are led:

```
                        (SUBSET (SET QS) (LEGALCARDS (QSO)))
3:14-35                 --- [by RULE342, analysis] --->
                        T
ASSUMING (SUIT-LED) = S
```

This assumption is used to simplify the first goal. First the goal is expanded:

```
(ACHIEVE (SUBSET (LEGALCARDS (QSO)) (SET QS)))

3:37-43 --- [by RULE6, analysis] --->

(UNTIL (PLAYED! QS)
       (ACHIEVE (AND [UNDO (LEGAL (QSO) C3)]
                     [IN C3 (DIFF (CARDS) (SET QS))])))
                          (LEGAL (QSO) C3)
```

```
      3:43                            --- [ELABORATE by RULE124] --->

                                      (AND [HAS (QSO) C3]
                                           [=> (LEADING (QSO))
                                               (OR [CAN-LEAD-HEARTS (QSO)]
                                                   [NEQ (SUIT-OF C3) H])]
                                           [=> (FOLLOWING (QSO))
                                               (OR [VOID (QSO) (SUIT-LED)]
                                                   [IN-SUIT C3 (SUIT-LED)])])
```

The assumption is applied by plugging in S as the value of (SUIT-LED):

```
                                               (OR [VOID (QSO) (SUIT-LED)]
                                                   [IN-SUIT C3 (SUIT-LED)])])

      3:62-79                                  --- [by RULE346, analysis] -->

                                               (IN-SUIT C3 S)
```

This substitution is performed by a rule discussed previously (§2.9):

RULE346: e -> c if e = c was assumed earlier

2.10.2. Modifying a Plan

Another technique for integrating two goals is

Construct a plan to achieve one goal and modify it to achieve the other.

This approach is illustrated in DERIV5, where the problem is to flush the Queen without taking it. The derivation starts by plugging in the plan derived in DERIV3 for flushing the Queen:

```
(AND [ACHIEVE (FLUSH QS)] [AVOID (TAKE ME QS) (TRICK)])

5:1 --- [REDUCE by RULE301] --->
(AND [UNTIL (PLAYED! QS) (ACHIEVE (LEAD-SPADE ME))]
     [AVOID (TAKE ME QS) (TRICK)])

5:2-4 --- [by RULE124, restructuring] --->
(UNTIL (PLAYED! QS)
       (ACHIEVE
          (AND [LEAD-SPADE ME]
               [NOT (DURING (TRICK) (TAKE ME QS))])))
```

Next "avoid taking the Queen" is reformulated as a condition on (CARD-OF ME):

```
5:5-31       --- [by analysis] --->
(UNTIL (PLAYED! QS)
       (ACHIEVE
          (AND [= (LEADER) ME]
               [SPADE! (CARD-OF ME)]
               [NOT (AND [IN QS (CARDS-PLAYED)]
                         [= (SUIT-OF (CARD-OF ME))
                            (SUIT-OF (CARD-OF (LEADER)))]
                         [FORALL X1
                            (CARDS-IN-SUIT-LED (CARDS-PLAYED))
                            (NOT (HIGHER X1 (CARD-OF ME)))])])))
```

The assumption `(= (LEADER) ME)` from the plan for `(FLUSH QS)` is plugged in using a general rule for propagating equalities:

RULE354: (and ... [= e c] ... [... e ...] ...) -> (and ... [= e c] ... [... c ...] ...)

The clause `[= (SUIT-OF (CARD-OF ME)) (SUIT-OF (CARD-OF ME))]` is eliminated. The quantified sub-expression is reduced to the sufficient condition `[HIGHER QS (CARD-OF ME)]` based on the assumption that `QS` will be in the suit led. This condition on `(CARD-OF ME)` is then used to constrain the action `(LEAD-SPADE ME)`:

```
5:32-49        --- [by RULE354, analysis] --->
(UNTIL (PLAYED! QS)
       (ACHIEVE
          (AND [= (LEADER) ME]
               [SPADE! (CARD-OF ME)]
               [HIGHER QS (CARD-OF ME)])))

5:52-53 --- [RECOGNIZE by RULE123] --->
(UNTIL (PLAYED! QS)
       (ACHIEVE (LEAD-SAFE-SPADE ME)))
```

Part of the derivation is a proof that underplaying the Queen is sufficient to avoid taking it, specifically, that the Queen will be in the suit led if it's played:

```
                           (IN QS (CARDS-PLAYED-IN-SUIT-LED))
5:40-45                    --- [by analysis] --->
                           (AND [IN QS (CARDS-PLAYED)]
                                [= S (SUIT-LED)])

                                (IN QS (CARDS-PLAYED))
5:42                            --- [REDUCE by RULE301] --->
                                T

                                (= S (SUIT-LED))
5:46-47                                --- [REDUCE by RULE301] --->
                                (= S S)

5:48                       --- [EVAL by RULE179] --->
                   (SHOW T)
```

The proof takes as premises that player ME will lead a spade and the Queen will be played in the trick. These premises are represented as follows:

```
(= (LEADER) ME)
(SPADE! (CARD-OF ME))
(IN QS (CARDS-PLAYED))
```

The proof shown contains a gap represented by the use of RULE301 at 5:46-47, where the suit led is determined to be spades. It could be filled in by expanding the definition of "suit led," propagating the first premise using RULE354 as before, reformulating the second premise as an equality, and propagating it:

```
                (SUIT-LED)
                --- [ELABORATE by RULE124] --->
                (SUIT-OF (LEADER))
                --- [by RULE354] --->
                (SUIT-OF (CARD-OF ME))

(SPADE! (CARD-OF ME))
--- [ELABORATE by RULE124] --->
(= (SUIT-OF (CARD-OF ME)) S)

                (SUIT-OF (CARD-OF ME))
                --- [by RULE354] --->
                S
```

2.10.3. Propagating Equalities

One theme apparent in this example is that reformulating constraints as equalities is a good idea, since they can readily be propagated in that form. FOO's transformational flavor makes such propagation a bit unwieldy, since inferences are generally made by transforming a single monolithic "*current expression*" rather than operating freely on a set of premises.

A more sophisticated problem-solving apparatus might maintain equivalence classes of expressions known to be equal [Nelson 79]. In this approach, the premises `(= (LEADER) ME)` and `(= (SUIT-OF (CARD-OF ME)) S)` would be represented in the graph structure shown in Figure 2-2. Each list expression and atom in the figure denotes a node in the graph. Each list expression is connected by an ordered list of arcs to the nodes representing its components, as denoted by the diagonal and horizontal lines in the figure. Expressions known to be equal are marked as belonging to the same equivalence class, as denoted by vertical lines from `(SUIT-LED)` to its definition `(SUIT-OF (CARD-OF (LEADER))`, from `(LEADER)` to `ME`, and from `(SUIT-OF (CARD-OF ME))` to `S`.

```
(SUIT-LED)
|\
= \--- SUIT-LED
|
(SUIT-OF (CARD-OF (LEADER)))            (SUIT-OF (CARD-OF ME))
 \                                      |\
  \--- SUIT-OF                          | \--- SUIT-OF
   \                                    |  \
    \--- (CARD-OF (LEADER))             |   \--- (CARD-OF ME)
          \                             =         \
           \--- CARD-OF                 |          \--- CARD-OF
            \                           |           \
             \--- (LEADER)              |            \--- ME
                  |\                    |
                  = \--- LEADER         S
                  |
                  ME
```

Figure 2-2: Graph Representation for Equality Propagation

The algorithm for propagating equalities puts two expressions in the same equivalence class if their corresponding components are equal. Thus (CARD-OF ME) and (CARD-OF (LEADER)) would get marked equal since their first components are the same atom and their second components are in the same equivalence class. Then (SUIT-OF (CARD-OF (LEADER)) and (SUIT-OF (CARD-OF ME)) would get marked equal for the same reason. The algorithm would automatically mark (SUIT-LED) equal to S, since it merges two equivalence classes whenever a member of one is marked equal to a member of the other.

Also, if two equal expressions have the same first component, *i.e.*, are both of the form (f ...) for some function f, and if f is known to be injective (one-to-one), the algorithm marks the corresponding components of the expressions equal. [Nelson 79] uses this method to deduce that if two CONS cells are equal, their respective CARs and CDRs are equal. This feature is not illustrated in the example above, but could be used, say, to deduce p = q from (CARD-OF p) = (CARD-OF q) since CARD-OF is an injective function, *i.e.*, a player only plays one card per trick.

2.10.4. Integrating Multiple Goals: Summary

When multiple goals interact, solving them independently may lead to inconsistencies. Approaches for constructing plans to achieve multiple goals include:

Record the assumptions made in solving one goal and apply them to the others.

Construct a plan to achieve one goal and modify it to achieve the other.

Representing assumptions as equalities of the form <expression> = <value> makes it easy to retrieve and apply them by substituting <value> for instances of <expression> elsewhere in the problem. Such a representation would permit application of an existing graph-based inference technique for efficient propagation of equalities.

2.11. Temporal Goals

An interesting issue in operationalization is how methods expressed in one paradigm can be brought to bear on a problem stated in another. For example, planning methods can be applied to the problem of making a sequence generator satisfy a specified constraint, by means of the following strategy:

1. *Reformulate* the constraint as a goal, with partial sequences as states, and sequence extension as the only action.
2. *Solve* the goal using planning rules.
3. *Reformulate* the constructed plan as a constraint on a left-to-right sequence generator that depends only on elements already generated.

2.11.1. Example: Satisfy Unique-Climax Constraint

DERIV14 uses this strategy to apply rules about achieving goals over time to the task of constructing a tone sequence (cantus firmus) satisfying various constraints. The constraint treated in DERIV14 is:

C4. The climax (highest note) of the cantus must occur exactly once.

This constraint is encoded as the expression

```
(= (#OCCURRENCES (CLIMAX CANTUS) CANTUS) 1)
```

Here CANTUS is a variable denoting the sequence to be constructed. The same constraint is incorporated in a heuristic search in DERIV13 (§3.5.3.4) by *choosing the climax a priori* (§2.12.1). In DERIV14, however, rules about goals are used to satisfy the constraint without determining the climax beforehand, by devising an obvious (to humans) strategy for choosing the next note:

> If the highest tone so far occurs only once, choose a lower tone to keep it that way; otherwise choose a higher tone to be the new (uniquely-occurring) climax.

DERIV14 begins by parameterizing the problem over "time," where each successive note corresponds to one time unit:

```
(ACHIEVE (= (#OCCURRENCES (CLIMAX CANTUS) CANTUS) 1))
14:1-4 --- [by RULE266] --->
(FORALL T1 (LB:UB 0 (- (# CANTUS) 1))
   ([ACHIEVE (= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1)] T1))
```

Here CANTUS-1 = (LAMBDA (I) (PREFIX CANTUS I)) denotes the subsequence of the cantus determined "so far," *i.e.*, at time I, and acts as a *fluent* (time-dependent quantity) inside the bracketed sub-expression. The overall expression says that the goal at each time T1 is for the highest note so far in the cantus to occur only once. Since this goal refers to the action to be taken at time T1 -- extending the cantus by the next note -- T1 ranges from zero up to one less than the length of the cantus. The expression specifies that this goal is to be achieved at *every* T1 in this range. This condition is unnecessarily strong, since all that matters is for the goal to be satisfied for the *last* value of T1. Representing the correct condition would require something like the idea of a deadline for achieving a goal. However, the interesting aspects of the derivation concern the goal itself, rather than the priority assigned to achieving it at any given time.

The idea of treating a sequence constraint as a temporal goal is implemented as

> RULE266: (achieve Ps) -> (forall t (lb:ub 0 (- (# s) 1)) ([achieve Ps'] t)),
> where s is a sequence to be constucted, and s' = (lambda (t) (prefix s t))

Achieving a temporal goal P_t is defined as *making it true at time* $t+1$. The functional NEXT takes a fluent -- implicit function of t -- and returns its next value, *i.e.*, the same implicit function applied to t+1. This provides the rationale for

> RULE282: (achieve P) -> (next P)

That is, P is achieved at time t if P is true at time t+1. The modal operator NEXT commutes freely with other functions:

(next (f e)) = (f (next e)) if e is a fluent (implicit function of time)

Of course

(next (f e)) = (f e) if e is time-independent

These two properties are combined in

RULE283: (next (f e_1 ... e_n)) -> (f e_1' ... e_n'),
where e_i' = (next e_i) if e_i is a fluent, and e_i' = e_i otherwise

As implemented, RULE283 assumes that any expression containing an unapplied function is a fluent. A more accurate method for mechanical identification of fluents would require marking primitive (undefined) concepts of the domain, like the LOC function, as fluents. An expression might then be identified as fluent or time-independent by expanding it in terms of primitive concepts and looking for unapplied fluents.

RULE282 and RULE283 justify the transformation

```
(ACHIEVE (= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1))
14:5-7 --- [by RULE282, RULE283] --->
(= (#OCCURRENCES (NEXT (CLIMAX CANTUS-1)) (NEXT CANTUS-1)) 1)
```

This expression specifies that extending the current partial sequence CANTUS-1 should yield a sequence with a unique climax. Further analysis depends on whether the climax changes in the process. This distinction is expressed by

RULE276: (P ... (next e) ...) ->
(or [and [not (change e)] [P ... e ...]] [and [change e] [P ... (next e) ...]])

Split a predicate on a fluent into two cases depending on whether the fluent changes.

RULE276 is applied with e = (CLIMAX CANTUS-1):

```
(= (#OCCURRENCES (NEXT (CLIMAX CANTUS-1)) (NEXT CANTUS-1)) 1)
14:8 --- [DISTRIBUTE by RULE276] --->
(OR [AND [NOT (CHANGE (CLIMAX CANTUS-1))]
         [= (#OCCURRENCES (CLIMAX CANTUS-1) (NEXT CANTUS-1)) 1]]
    [AND [CHANGE (CLIMAX CANTUS-1)]
         [= (#OCCURRENCES (NEXT (CLIMAX CANTUS-1))
                          (NEXT CANTUS-1))
            1]])
```

The first case, where the climax doesn't change, is:

```
(AND [NOT (CHANGE (CLIMAX CANTUS-1))]
     [= (#OCCURRENCES (CLIMAX CANTUS-1) (NEXT CANTUS-1)) 1])
```

The object is to reduce it to a condition on the next note to be chosen. This requires the transformation

```
                                  (NEXT CANTUS-1)
14:10-17                          --- [by analysis] --->
                                  (APPEND CANTUS-1 (LIST (NEXT NOTE)))
```

This transformation falls out quite neatly from the definition of the functional

```
NEXT = (LAMBDA (F)
          (LAMBDA (I) (F (+ I 1))))
```

It requires the segmentation of $s_1 \ldots s_{i+1}$ into $s_1 \ldots s_i$ followed by s_{i+1}, effected by

RULE274: (prefix s (+ i 1)) -> (append (prefix s i) (list (nth s (+i 1))))

Finally, functions are freely treated as functionals, as expressed by

RULE393: (lambda (t) (f $e_1 \ldots e_n$)) -> (f (lambda (t) e_1) ... (lambda (t) e_n))

The rest of the analysis of the first case is fairly straightforward:

```
(= (#OCCURRENCES (CLIMAX CANTUS-1)
                 (APPEND CANTUS-1 (LIST (NEXT NOTE))))
   1)

14:18 --- [DISTRIBUTE by RULE284] --->

(= (+ (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1)
      (#OCCURRENCES (CLIMAX CANTUS-1) (LIST (NEXT NOTE))))
   1)
```

The second term of the sum is 1 if the next note repeats the climax, 0 otherwise, so the first term must be 0 or 1:

```
14:19-27 --- [by analysis] --->
(IF (= (NEXT NOTE) (CLIMAX CANTUS-1))
    (= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 0)
    (= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1))
```

It can't be 0, since (CLIMAX CANTUS-1) is by definition an element of CANTUS-1. So it must be 1, and the next note can't repeat the climax:

```
14:28-36 --- [by case elimination] --->
(AND [= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1]
     [NOT (= (NEXT NOTE) (CLIMAX CANTUS-1))])
```

Nor can the next note be higher than the climax:

```
        (NOT (CHANGE (CLIMAX CANTUS-1)))
    14:40-45 --- [BY RULE277, analysis] --->
        (NOT (HIGHER (NEXT NOTE) (CLIMAX CANTUS-1)))
```

The rationale for this step is a general principle:

The extremum of an expanding set changes iff one of the new elements lies beyond it.

This is expressed by

RULE277: (change (Max$_R$ s)) -> (R (Max$_R$ (change s)) (Max$_R$ s)) if s is expanding, where (Max$_R$ s) is the maximal element of s with respect to the ordering R

2.11.1.1 Representational Ambiguity and Conciseness

The construct (change e) is used ambiguously to mean both Δe and $\Delta e \neq 0$. Thus (change s) denotes the new elements of the expanded set or sequence s', while (change (Max$_R$ s)) denotes the proposition that the maximum changes, *i.e.*, that (Max$_R$ s) $\neq$ (Max$_R$ s'). Such ambiguity is representationally sloppy but concise. In that respect it resembles the practice of quantifying over sequences as though they were sets. The danger of this practice is illustrated by representing the number of occurrences of C in the sequence S as

```
    (# (SET-OF X S (= X C)))
```

The value of this expression should always be either 0 or 1. A precise but longer notation for number of occurrences of C in a sequence S is

```
    (# (SET-OF X S (= (VALUE-OF X) C)))
```

Here a sequence S = $x_1 \ldots x_n$ is represented as a set of pairs <i, x_i>, and (value-of <i, x_i>) = x_i.

Anyway, the case in which the climax doesn't change reduces to

```
    (AND [= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1]
         [NOT (= (NEXT NOTE) (CLIMAX CANTUS-1))]
         [NOT (HIGHER (NEXT NOTE) (CLIMAX CANTUS-1))])

    14:47-50 --- [by analysis] --->

    (AND [= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1]
         [LOWER (NEXT NOTE) (CLIMAX CANTUS-1)])
```

The case in which the climax does change is

```
(AND [CHANGE (CLIMAX CANTUS-1)]
     [= (#OCCURRENCES (NEXT (CLIMAX CANTUS-1)) (NEXT CANTUS-1))
        1])
```

The climax changes iff the new note is higher than the existing climax:

```
14:52-60 --- [by analysis] --->
(AND [HIGHER (NEXT NOTE) (HIGHEST CANTUS-1)]
     [= (#OCCURRENCES (NEXT NOTE) (NEXT CANTUS-1)) 1])
```

As in the first case, the expression (#OCCURRENCES ...) is split into two terms:

```
14:62-68 --- [by analysis] --->
(AND [HIGHER (NEXT NOTE) (HIGHEST CANTUS-1)]
     [= (+ (#OCCURRENCES (NEXT NOTE) CANTUS-1) 1) 1])

14:69 --- [by RULE290] --->
(AND [HIGHER (NEXT NOTE) (HIGHEST CANTUS-1)]
     [= (#OCCURRENCES (NEXT NOTE) CANTUS-1) 0])
```

The second conjunct is subsumed by the first:

```
     (= (#OCCURRENCES (NEXT NOTE) CANTUS-1) 0)
14:70-72 --- [by analysis] --->
     (NOT (IN (NEXT NOTE) CANTUS-1))
14:73-77 --- [by RULE304, analysis] --->
     T
```

The idea here is that

An entity that lies beyond an extreme element of a set can't be a member of it.

This idea is expressed by

RULE304: (in x s) -> nil if (R x (Max_R s)) for some ordering R on s

In this case, x = (NEXT NOTE), s = CANTUS-1, R = HIGHER, and Max_R = HIGHEST.

Thus the two cases reduce to

```
(OR [AND [= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1]
         [LOWER (NEXT NOTE) (CLIMAX CANTUS-1)]]
    [HIGHER (NEXT NOTE) (CLIMAX CANTUS-1)])
```

The derivation ends by translating this expression, which contains the fluent CANTUS-1, back into a static sequence constraint. This is accomplished by plugging in the quantifier variable T1 to which the expression is functionally applied:

```
(FORALL T1 (LB:UB 0 (- (# CANTUS) 1))
   ([OR [AND [= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1]
             [LOWER (NEXT NOTE) (CLIMAX CANTUS-1)]]
        [HIGHER (NEXT NOTE) (CLIMAX CANTUS-1)]]
    T1))
14:82-87 --- [by RULE273, analysis] --->
(FORALL T1 (LB:UB 0 (- (# CANTUS) 1))
   (OR [AND [= (#OCCURRENCES (CLIMAX (CANTUS-1 T1)) (CANTUS-1 T1))
               1]
            [LOWER (NOTE (+ T1 1)) (CLIMAX (CANTUS-1 T1))]]
       [HIGHER (NOTE (+ T1 1)) (CLIMAX (CANTUS-1 T1))]))
```

This transformation is performed by

RULE273: ([f ... g_1 ... g_n ...] t) -> (f ... (g_1 t) ... (g_n t) ...), where g_i are fluents

The resulting expression could be used to constrain a left-to-right tone sequence generator, since the quantified condition on `(NOTE (+ T1 1))` depends only on `(CANTUS-1 T1)`, *defined* as `(PREFIX CANTUS T1)`.

2.11.2. Temporal Goals: Summary

A general strategy for applying a method to a problem is:

1. *Translate* the problem into the language of the method.
2. *Solve* the translated problem using the method.
3. *Translate* the solution back into the original language of the problem, if necessary.

In particular, planning methods can be applied to sequence constraints:

1. *Translate* the sequence constraint into a temporal goal.
2. *Solve* the temporal goal using planning methods.
3. *Translate* the plan back into an operational constraint on a left-to-right sequence generator.

In the example treated in the section, the planning methods split goals into different cases and simplify them based on general knowledge about extrema of sets and change over time.

2.12. Parameterization

An operationalization method may reduce a problem rather than solve it outright. One such is:

Assume the value of some problem feature is determined a priori.

This value might be any of several things:

1. A design parameter chosen as part of the operationalization process.
2. An input variable in a parameterized solution.
3. A quantity known at the time the solution is used but not at the time the problem is solved.

As it happens, all the illustrations of this technique drawn from the derivations occur in the context of applying the heuristic search method and are described in more detail in the discussion of that method (§3.5). However, the ideas involved can be discussed independently of the details of heuristic search.

2.12.1. Parameterizing a Problem

The problem of satisfying a constraint on some feature of a constructed object can often be simplified by first deciding the value of that feature. For example, consider the problem of constructing a cantus (sequence of musical tones) satisfying the constraint

C4. The climax tone of the cantus must occur exactly once.

This problem can be simplified by choosing the climax tone *a priori*, and then using it exactly once in the cantus, rather than determining the climax dynamically as the cantus is generated, as in DERIV14 (§2.11).

The moral of this example is, "To simplify a problem, parameterize it." More precisely,

To constrain a constructed object to satisfy a specified property, first choose the value of some parameter on which the property depends, and then construct the object so as to ensure that (a) the parameter takes on the chosen value, and (b) the property is satisfied for that value.

Ideally, it should be possible to satisfy (a) and (b) by constraining the construction process in a way that at each point depends only on the part constructed so far. This property is important in heuristic search (§3), where the search space is pruned by applying constraints to incomplete paths.

2.12.1.1 Parameterizing the Cantus Climax

The strategy of reducing a problem by parameterizing it is illustrated in the following step from DERIV13 (shown here with some cosmetic changes for compatibility with DERIV14):

```
(ACHIEVE (= (#OCCURRENCES (CLIMAX CANTUS) CANTUS; 1))
13:58     --- [RESTRICT by RULE256] --->
(ACHIEVE (AND [= (CLIMAX CANTUS) CLIMAX1]
              [= (#OCCURRENCES CLIMAX1 CANTUS) 1]))

(ASSUME (SELECT CLIMAX1 (TONES)))
```

This transformation is suggested by

RULE256: (P ... (f s) ...) -> (and [= (f s) c] [P ... c ...]),
where c is to be selected from (range f) before s is constructed

The first conjunct constrains the climax of the cantus to be the pre-selected tone:

```
                (= (CLIMAX CANTUS) CLIMAX1)
13:60-62        --- [by analysis] --->
                (AND [IN CLIMAX1 CANTUS]
                     [FORALL X1 CANTUS (NOT (HIGHER X1 CLIMAX1))])
```

Thus the cantus must contain the pre-selected tone and no higher tones.

The second conjunct constrains the pre-selected tone to occur exactly once in the cantus:

```
                (= (#OCCURRENCES CLIMAX1 CANTUS) 1)

13:83           --- [ELABORATE by RULE340] --->
                (AND [>= (#OCCURRENCES CLIMAX1 CANTUS) 1]
                     [=< (#OCCURRENCES CLIMAX1 CANTUS) 1])

13:84-116       --- [by analysis] --->
                (AND [IN CLIMAX1 CANTUS]
                     [FORALL I1 (LB:UB 0 (- (# CANTUS) 1))
                        (OR [NOT (= (NOTE (+ I1 1)) CLIMAX1)]
                            [NOT (IN CLIMAX1 (PREFIX CANTUS I1))])])
```

That is, the pre-selected tone must be used at some point in the generation of the cantus, but cannot be used thereafter. The reformulated conjuncts can be used to constrain a left-to-right tone sequence generator, since they depend only on the notes generated so far at each point. The constraint `(IN CLIMAX1 CANTUS)` is satisfied as soon as the pre-selected tone is generated; this is explained more precisely in the discussion of heuristic search (§3.5.5.3).

2.12.1.2 Parameterizing the Cantus Length

The same rule for parameterization could also be used to simplify another constraint:

C1. The cantus should be between 8 and 16 notes long.

RULE256 suggests choosing the cantus length *a priori*:

```
(ACHIEVE (IN (# CANTUS) (LB:UB 8 16)))
--- [RESTRICT by RULE256, analysis] --->
(ACHIEVE (= (# CANTUS) LENGTH1))

(ASSUME (SELECT LENGTH1 (LB:UB 8 16)))
```

This expression could be used to produce a left-to-right generator of the form

```
Select LENGTH1 from (LB:UB 8 16) -- at random or by asking user

For I1 from 1 to LENGTH1 generate (NOTE I1)
```

It's interesting to note that Meehan's cantus generation program [Meehan 72] has this form.

2.12.1.3 Parameterizing the Cantus Range

The same rule could be used to simplify the constraint

C3. The melodic range of the cantus should not exceed a major tenth.

Here, RULE256 suggests choosing the lowest and highest notes *a priori*:

```
(ACHIEVE (=< (MELODIC-RANGE CANTUS) (MAJOR 10)))

--- [ELABORATE by RULE124] --->
(ACHIEVE (=< (SIZE (INTERVAL (LOWEST CANTUS) (HIGHEST CANTUS)))
             (MAJOR 10)))

--- [RESTRICT by RULE256] --->
(ACHIEVE (AND [=< (SIZE (INTERVAL LOWEST1 HIGHEST1)) (MAJOR 10)]
              [= (LOWEST CANTUS) LOWEST1]
              [= (HIGHEST CANTUS) HIGHEST1])

(ASSUME (SELECT LOWEST1 (TONES)))
(ASSUME (SELECT HIGHEST1 (TONES)))

--- [by analysis] --->
(ACHIEVE (AND [=< (SIZE (INTERVAL LOWEST1 HIGHEST1)) (MAJOR 10)]
              [IN LOWEST1 CANTUS]
              [IN HIGHEST1 CANTUS]
              [FORALL I1 (LB:UB 1 (# CANTUS))
                 (AND [NOT (LOWER (NOTE I1) LOWEST1)]
                      [NOT (HIGHER (NOTE I1) HIGHEST1)])]))
```

The first conjunct constrains the choice of LOWEST1 and HIGHEST1, and can be satisfied by choosing them appropriately (§2.7.1). The last conjunct constrains the choice of each note independent of the other notes, and can therefore be incorporated in a left-to-right generator. This solution resembles Meehan's program, which asks the user to specify a climax tone (used as the highest note of the cantus) and a lowest permissible tone (not always included in the cantus). The second and third conjuncts constrain the actual melodic range of the cantus to cover the full interval between the pre-selected tones, and are not required by the original constraint that the range be less than a major tenth. A derivation of a more sophisticated solution omitting these unnecessary constraints is sketched later (§3.5.5.5).

2.12.2. Parameterizing the Solution

Sometimes a problem can be simplified a good deal just by assuming that some piece of information will be known at the time the solution is used. This idea is illustrated in DERIV2, where "avoid taking points" is operationalized as a heuristic search (§3). The search space consists of the possible scenarios (card sequences) for the trick; the purpose of the search is to determine whether any of these scenarios cause player ME to take points. The search can be simplified considerably by assuming that the suit led, the cards played by player ME's predecessors, and the card player ME is considering playing are all known before the search starts. The details of the assumption process are given later (§3.4.2). The general idea underlying it is

To simplify a problem, parameterize the solution.

More precisely:

To evaluate a property $P_{(f\ c)}$ of some object c, choose some feature (f c) on which the property depends, make it a variable v, and express the property as an evaluable function P_v

If the value of the feature (f c) is known at evaluation time, it can be assigned to the variable v before the expression is evaluated; otherwise, the expression P_v can be treated as a function.

An example of a feature known at evaluation time is the card sequence played by player ME's predecessors (assuming that the search is performed as part of the process whereby player ME chooses a card to play).

An example of a feature not determined at evaluation time is the card to be played by player ME. If this is made a variable, player ME can evaluate each card it's considering playing by executing the search procedure with that card as the value of the variable.

An example of a feature sometimes but not always determined at evaluation time is the suit led. If player ME is leading, the suit led is unknown, and player ME can execute different searches with different values for the "suit led" variable corresponding to the respective suits of the different cards it's considering leading. If someone else is leading, the suit led will be known at the time of player ME's turn, and can be assigned to the "suit led" variable before the search begins.

2.12.3. Parameterization: Summary

Parameterization reduces a problem $P_{(f \ldots)}$ to a simpler problem P_v. There are several cases:

1. The value of (f ...) will be known or chosen at evaluation time and can be assigned to the variable v.
2. P_v will be evaluated for different values of (f ...).
3. The problem of ensuring (f ...) = v is treated as an additional goal.

This simple strategy accounts for several design decisions in Meehan's cantus generating program.

2.13. Operationalization Mthods: Summary

The term "operationalization" refers to the process of making an expression operational, *i.e.*, "operational-ization." The alternative segmentation "operation-alization" suggests another definition: reformulating an expression in terms of a given set of executable *operations*.[6] This view is based on the task agent model shown in Figure 1-1 (§1.1.1). In this model, an expression can be non-operational for any of several reasons. An expression to be evaluated might be defined in terms of *unobservable* data, or depend on an *unpredictable* future event. Similarly, a goal to be achieved might not be expressed as a *doable* action, or it might depend on an *uncontrollable* choice made by some other agent.

Each operationalization method presented in this chapter can be characterized as overcoming a particular kind of obstacle to operationality:

1. The pigeon-hole method reformulates a function of *unobservable* data in terms of *known* data (§2.2).

[6]Contrary to some colleagues' suggestions, "operationalization" does not mean rationalizing -- or nationalizing -- opera.

2. The disjoint subsets method estimates a function of *unobservable* data based on *known* data (§2.3).

3. Historical reasoning computes a function of *unobservable* data based on *remembered* data (§2.4).

4. The set depletion method helps translate a *non-operational* goal into a series of *doable* actions (§2.5).

5. Finding a necessary or sufficient condition (§2.6) helps:

   Evaluate (in some situations) a function of *unobservable* data based on *known* data (§2.6.2) (§2.6.3).

   Reformulate a *non-operational* goal into a *doable* action that achieves a sufficient condition (§2.6).

6. FOO's model of choice (§2.7) is used in both planning and evaluation:

   Constrained choice reformulates a goal involving an *unpredictable* choice as a *doable* plan (§2.7.1).

   Forcing helps translate a goal involving an *uncontrollable* choice into a *doable* plan (§2.7.2).

   Pessimism estimates a function of an *unpredictable* choice based on *known* data (§2.7.3).

7. Dependence analysis estimates a function of *unobservable* data based on *known* data (§2.8).

8. Simplifying assumptions (§2.9) help:

   Create a *doable* plan to achieve a goal involving an *unpredictable* condition (§2.9.2).

   Estimate a function of *unobservable* data based on an untested assumption (§2.9.3).

9. Integrating multiple goals helps translate a *non-operational* conjunction into a *doable* plan (§2.10).

10. Temporal planning helps translate a goal involving an *unpredictable* sequence into a *doable* plan (§2.11).

11. Parameterization makes an *unpredictable* quantity into a choice variable (§2.12).

12. Choice set ordering reformulates a *non-operational* goal in terms of a *controllable* choice (§2.8.3).

13. Heuristic search evaluates a function of *unpredictable* data by considering alternative possibilities (§3).

14. Reformulation (§4) helps:

    Evaluate a function of *unobservable* data based on *observable* data (§4.3.1.1).

    Translate a *non-operational* goal into a *controllable* choice (§4.3.1.2).

    Translate a *non-operational* goal into a *doable* action (§4.3.1.3).

The last items on this list refer to methods not yet discussed in detail. *Reformulation* sometimes suffices to solve an operationalization problem, but more often serves to enable the application of one of the operationalization methods described in this chapter. This crucial process is discussed in (§4).

Heuristic search (§3) does not fit the simplistic model of agent-environment interaction depicted in Figure 1-1. For example, this model ignores computational costs, which are a central factor motivating the search refinement rules discussed in (§3.4). Also, heuristic search is a complex process internal to the agent, but the model focuses on interactions with the external environment. Heuristic search can be embedded in the model to some extent by treating it as an external process occurring over time: for example, the complete solution path can be treated as "unpredictable" data, and desired properties of a solution can be reformulated as tests on "observable" partial paths or "controllable" individual choice points. However, this view fails to represent the way such tests (internally computed functions) are used in the search (external environmental process).

Another consequence of the complexity of heuristic search relative to the rather simple methods discussed so far is the complexity of the process whereby a problem is reformulated as a heuristic search problem. This process is a fundamental activity in the practice of AI or "knowledge engineering," and serves as the focus of the next chapter.

Chapter 3
Using the Heuristic Search Method

When AI researchers design intelligent programs, they draw upon a repertoire of general operationalization techniques, including the "weak methods" of generate-and-test, hill-climbing, heuristic search, matching, and means-end analysis abstractly formalized by Newell as data-flow graphs [Newell 69]. An important skill in designing such programs is the ability to take a problem expressed in the language of a particular task domain and operationalize it in terms of one of these methods.

This process can be viewed as mapping the problem into a call on a general procedure for an AI method (heuristic search, for example) by finding suitable values for the arguments (which may include generators, tests, and orderings for controlling the search). This is essentially what it means to formulate a problem like "avoid taking points" or "compose a *cantus firmus*" in terms of a heuristic search, and relatively little has been accomplished toward getting computers to do it automatically. Moore encoded Newell's description of heuristic search as a MERLIN schema [Moore 71] and instantiated it to represent the Logic Theorist [Newell 57] program. The resulting structure was operational in that MERLIN could prove theorems by executing it, but the instantiation *process* by which this structure was constructed -- *i.e.*, the derivation of the appropriate generators and tests from the LT specifications -- was carried out by hand.

Perhaps the most advanced effort so far toward automatic application of AI techniques is the UNDERSTAND program [Simon 77], which reads an English description of the Tower of Hanoi problem and operationalizes it as a means-end analysis problem by constructing an appropriate state space representation and associated operators.

Mechanizing the application of an AI method raises several questions:

How to *represent* a general method so that it can be reasoned about mechanically

How to *map* a particular problem onto the general problem statement for the method

How to *fill in* information required by the method but not explicit in the original problem

How to *refine* the solution by using additional knowledge about the problem domain

This chapter explores these questions for the heuristic search method (hereafter abbreviated "HSM"). It presents some concrete answers in the form of a schematic representation of HSM and some rules that operate on it, and illustrates them by showing how they are used to operationalize the Hearts advice "avoid taking points." The generality of the formulation is tested by applying it to a simplified music composition task.

3.1. A Slightly Non-Standard Version of Heuristic Search

Newell [Newell 69] described the general problem statement for HSM as follows:

Given:	a set $\{x\}$, the problem space;
	a set of operators $\{q\}$ with range and domain in $\{x\}$;
	an initial element, x_0;
	a desired element, x_d;
Find:	a sequence of operators, $q_1, q_2, \ldots, q_n$, such that they transform x_0 into x_d;

$$q_n[q_{n-1}[\ldots q_1(x_0)\ldots]] = x_d$$

He mentioned some variations:

Initially, a set of elements may be given, rather than just one, with the search able to start from any of them. Likewise, finding any one of a set of elements can be desired, rather than just finding a single one. This final element (or set) can be given by a test instead of by an element ... the operators need not involve only a single input and a single output.

He described the heuristic search procedure as follows:

The *initial element* x_0 is the initial current position; operators are selected and applied to it; each new element is compared with x_d to see whether the problem is solved; if not, it is added to a list of obtained positions (also called the "try list" or the "subproblem list"); and one of these positions is selected from which to continue the search.... The search is guided (i.e., the tree pruned) by appropriate selection and rejection of operators and elements.

FOO uses a somewhat different formulation of HSM in which everything is expressed in terms of (operator) sequences, called *paths*. The search procedure extends a path by explicitly appending an operator to it; the fact that operators map one state to another is implicit in the tests applied to paths, *i.e.*, operators are not functionally applied to states. This procedure can be described by modifying Newell's description (differences are underlined):

The null path is the initial current position; choice elements are selected and appended to it; each new path is tested to see whether the problem is solved; if not, it is added to a list of paths; and one of these paths is selected from which to continue the search. The search is guided by appropriate selection and rejection of choice elements and paths.

The choice elements need not be operators; this procedure can be applied to any problem of the form

Find a sequence of choices satisfying a given criterion.

Thus the first step in operationalizing a problem in terms of HSM is to identify the sequence of choice points involved and the set of admissible alternatives at each one, and to reformulate the solution criterion as a computable predicate on the choice sequence. This provides enough information to specify an executable but inefficient (*i.e.*, combinatorially explosive) *generate-and-test* search procedure, shown in Figure 3-1 as a data-flow graph. Such a procedure can then be refined into a genuine heuristic search, shown in Figure 3-2, by reorganizing the search process so that constraints on the overall solution are applied as early as possible in the search: to order or reject paths, to order or filter the generation of alternatives at each choice point, and even to reduce the depth or breadth of the search space itself.

3.1.1. Overview

The remainder of the chapter is organized as follows. (§3.2) describes a generic heuristic search procedure and a corresponding schematic representation of HSM. Slots in the schema correspond to problem-specific components of this procedure, such as the tests used to prune paths. (§3.3) illustrates the mechanical instantiation of the HSM schema for the "avoid taking points" example, and describes the initial "naive" (inefficient) search it yields. (§3.4) shows how this solution can be iteratively refined using knowledge about the domain and analysis of the search criterion. (§3.5) applies the whole process to an example from the domain of music composition, and (§3.6) draws some conclusions about the generality of the approach.

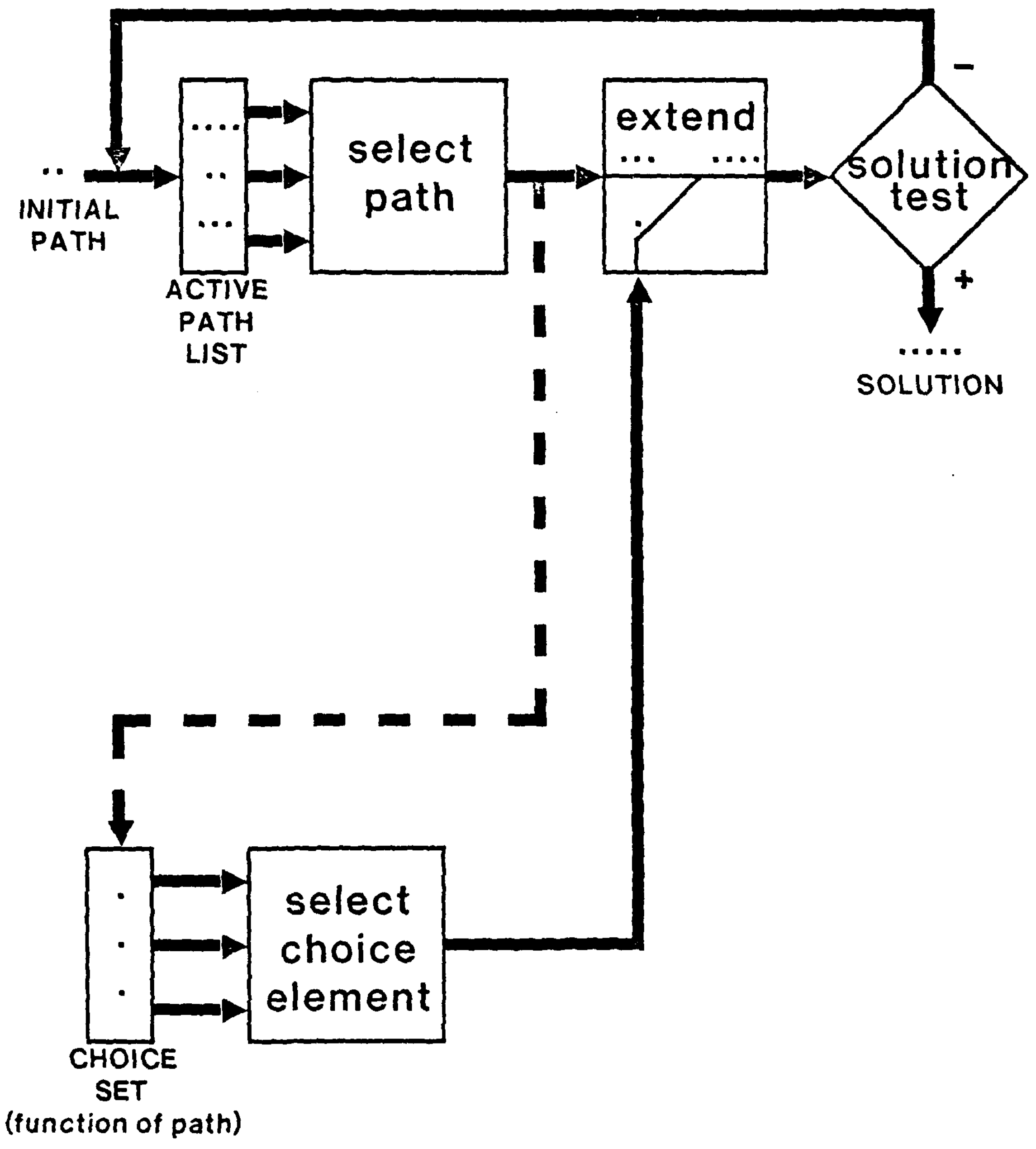

Figure 3-1: Generate-and-Test Procedure

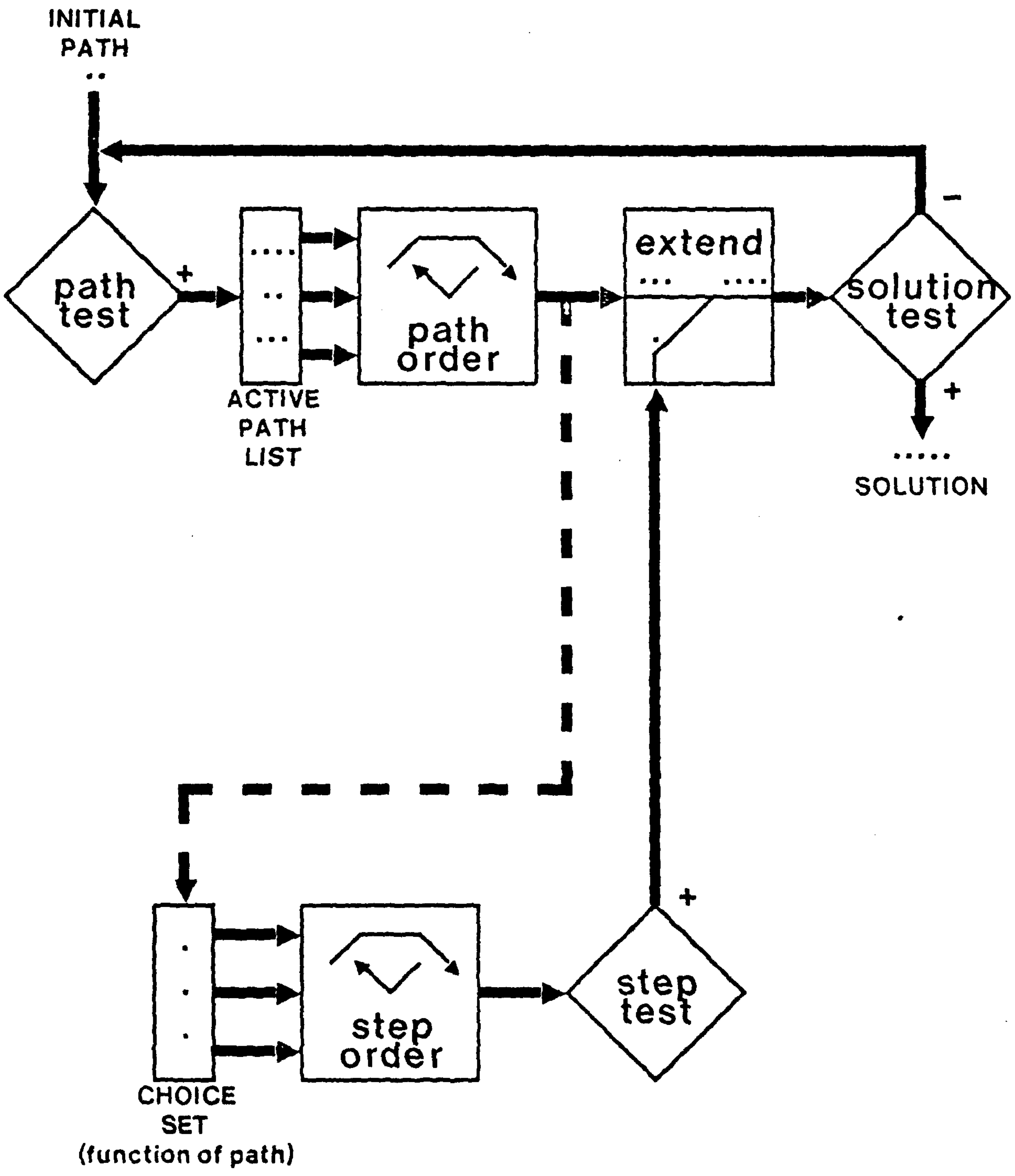

Figure 3-2: Generic Heuristic Search Procedure

3.2. A Schematic Representation of HSM

According to the slightly non-standard formulation of HSM used in FOO, its purpose is to

Find a sequence satisfying a given criterion.

Thus the search space for HSM contains partial sequences or *paths*. The search proceeds by selecting from a list of paths under consideration, extending the chosen path by an element chosen from a set of alternatives, and testing if the extended path satisfies the search criterion. If so, the search halts successfully; otherwise, the path is added to the active path list. A path is removed from the active list if all its extensions have been considered or it can be determined not to lead to a solution. This cycle repeats until a solution is found or the list is exhausted.

This rough description provides enough context to describe the key components of FOO's HSM schema.

The search starts with a single path, the **initial-path**.

The alternative extensions for a path are given by the **choice-set** function. Since the range of choices may vary at different points in the search, this function takes as its argument an index that identifies a particular choice point.

The order in which alternative extensions are generated is controlled by a **step-order**, a predicate on sequence elements. Elements satisfying this predicate are considered before others. The step-order is represented as a unary predicate rather than a binary ordering relation for simplicity. An arbitrary ordering on the choice set could be expressed as a fuzzy unary predicate: extensions that satisfied the predicate relatively well would be considered before those satisfying it to a lesser degree.

The set of extensions generated for a given path is filtered by a **step-test** predicate: thus the extensions to a given path are enumerated by a generate-and-test process, with the choice-set function as the generator and the step-test predicate as the test. Since the step-test and step-order predicates may depend on the path being extended, they take a path index as a second argument.

The order in which paths are selected for extension is controlled by a **path-order** predicate on paths; paths satisfying this predicate are considered first.

A newly generated path must satisfy a **path-test** in order to be added to the active list.

A solution path must satisfy both a **solution-test** based on the search criterion, and a **completion-test** that checks if the path covers the complete sequence of choice points.

The generic heuristic search procedure shown in Figure 3-2 can now be specified more precisely:

1. The first path is the **initial-path**.
2. If the path satisfies the **path-test**, insert it in the active path list.

3. Choose a path from the active list, preferably one that satisfies the **path-order**. If the list is empty, the search has failed.
4. Choose a previously untried choice element from the choice-set, preferably satisfying the **step-order**. If all extensions of the path have been tried, remove it from the active list and go back to step 3.
5. If the choice element fails the **step-test**, go back to step 4 and try again.
6. Extend the path by appending the choice element to it. If the extended path satisfies the **solution-test** and **completion-test**, it's a solution. Otherwise go to step 2.

I have not actually implemented this procedure but see no conceptual obstacles to doing so. Execution of the procedure would require the ability to evaluate expressions defined in terms of the runtime task environment, *e.g.*, `(CARDS-IN-HAND ME)` and `(SUIT-LED)`. The focus of this chapter is not on the procedure itself, but on the process by which a problem is operationalized in terms of an appropriate call on the procedure.

3.3. Instantiating the HSM Schema for a Given Problem

The purpose of HSM is to

Find a sequence of choices satisfying a given criterion.

There are several steps involved in mapping a particular problem like "avoid taking points" onto this general problem statement:

1. *Reformulate* the problem to make it possible to
2. *Recognize* the problem as potentially amenable to HSM; then
3. Identify the *choice sequence* for the problem and
4. Express the *search criterion* as a function of the choice sequence.

Given the choice sequence and search criterion for the problem, the HSM schema can be instantiated to formulate a crude but executable search.

3.3.1. Mapping a Problem onto the HSM Problem Statement

The first step in mapping "avoid taking points" onto the HSM problem statement is to reformulate it so the applicability of HSM becomes manifest, as illustrated in the following excerpt from DERIV2, the derivation for the "avoid taking points" example:

```
(AVOID-TAKING-POINTS (TRICK))

2:1 --- [ELABORATE by RULE124] --->
(AVOID (TAKE-POINTS ME) (TRICK))

2:2 --- [ELABORATE by RULE124] --->
(ACHIEVE (NOT (DURING (TRICK) (TAKE-POINTS ME))))
```

The relevance of HSM to the reformulated problem is indicated by a general rule:

Use heuristic search to evaluate a predicate on a scenario in which choices are made.

In the case at hand, (TRICK) is the scenario in which choices are made. In FOO, the rule is

```
RULE306: (P ... e ...)
->(HSM with   (problem : (P ... e ...))
              (object : e)
              (choice-seq : (choice-seq-of e)))
if e contains an event sequence with choices in it
```

In RULE306, the names "problem," "object," and "choice-seq" denote components of the HSM schema: respectively, the expression to be evaluated, the scenario referred to in the expression, and the sequence of choice points in the scenario.

FOO recognizes that (DURING (TRICK) (TAKE-POINTS ME)) is a predicate on the scenario

```
(TRICK) = (SCENARIO (EACH P (PLAYERS) (PLAY-CARD P))
                    (TAKE-TRICK (TRICK-WINNER)))
```

The sequence (EACH P (PLAYERS) (PLAY-CARD P)) in this scenario contains the event

```
(PLAY-CARD P) = (CHOOSE C (LEGALCARDS P) (PLAY P C))
```

This enables FOO to identify the expression (DURING (TRICK) (TAKE-POINTS ME)) as a predicate on a scenario in which choices are made. RULE306 suggests using HSM to evaluate this expression, *i.e.*, to determine whether there is some sequence of choices satisfying the predicate. Applying RULE306 produces a partial instantiation of the HSM schema:

```
             (DURING (TRICK) (TAKE-POINTS ME))
2:3          --- [by RULE306] --->
             HSM1
(HSM1 <- PROBLEM : (DURING (TRICK) (TAKE-POINTS ME)))
(HSM1 <- OBJECT : (TRICK))
(HSM1 <- CHOICE-SEQ : (CHOICE-SEQ-OF (TRICK)))
```

The notation above reflects the fact that this step is not simply a transformation from the expression (DURING (TRICK) (TAKE-POINTS ME)) to the symbol HSM1: in addition, values are filled in for three of HSM1's components, as indicated by the parenthetical assignments shown after the transformation. As LISP users might expect, such assignments are implemented in FOO by putting component-value pairs on the property list of the atom HSM1.

3.3.2. Finding the Sequence of Choices that Affect the Predicate

The next step is to analyze the definition of TRICK to identify the sequence of choices involved:

```
                    (CHOICE-SEQ-OF (TRICK)))
2:5-13              --- [RULE307-309, analysis] --->
                    (EACH P1 (PLAYERS)
                         (CHOOSE (CARD-OF P1)
                              (LEGALCARDS P1)
                              (PLAY P1 (CARD-OF P1))))
```

The CHOICE-SEQ-OF function projects an event sequence onto the subsequence of events that are choice events. It is computed by expanding the definition of TRICK and applying some rules for extracting choice sequences from event descriptions:

RULE307: (choice-seq-of (each x S Ex)) -> (apply append (each x S (choice-seq-of Ex)))

RULE308: (choice-seq-of s) -> nil if s involves no choice

RULE309: (choice-seq-of (choose x S Ex)) -> (list (choose x S Ex))

Several components of the HSM schema can now be instantiated:

```
2:15 --- [ELABORATE by RULE311] --->
(HSM1 <- SEQUENCE : CARD-OF)
(HSM1 <- CHOICES : (PROJECT CARD-OF (PLAYERS)))
(HSM1 <- CHOICE-SETS : (LAMBDA (P1) (LEGALCARDS P1)))
(HSM1 <- INDICES : (PLAYERS))
(HSM1 <- INDEX : P1)
(HSM1 <- VARIABLES : NIL)
(HSM1 <- BINDINGS : NIL)
(HSM1 <- INITIAL-PATH : NIL)
(HSM1 <- COMPLETION-TEST :
          (LAMBDA (PATH) (= (# PATH) (# (PLAYERS)))))
```

This is done by extracting them from the choice sequence description or using default values:

```
RULE311:    (HSM with    choice-seq : (each x indices (choose (f x) S A))))
->          (HSM with    (sequence : f) -- function from index to choice point
                         (choices : (project f indices)) -- sequence of chosen objects
                         (choice-sets : (lambda (x) S)) -- choice set at choice point x
                         (indices : indices) -- choice point indexing sequence
                         (index : x) -- choice point index variable
                         (variables : nil) -- (initially empty) list of temporary variables
                         (bindings : nil) -- corresponding list of values
                         (initial-path : nil) -- default start state for search
                         (completion-test :
                               (lambda (path) ( = (# path) (# indices)))))
                               -- test if path covers entire choice sequence
```

Some of these components, *e.g.*, CHOICES, are not directly used by the search procedure, but are used by rules that instantiate other components.

3.3.3. Reformulating the Search Criterion in terms of the Choice Sequence

At this point the card sequence for the trick has been identified as the choice sequence in the HSM problem statement

Find a sequence of choices satisfying a given criterion.

Accordingly, the value of the CHOICES component in the HSM schema has been filled in as (PROJECT CARD-OF (PLAYERS)), otherwise known as (CARDS-PLAYED). It remains to reformulate the search criterion as an evaluable predicate on this sequence, *i.e.*, to solve the problem:

```
(REFORMULATE (DURING (TRICK) (TAKE-POINTS ME))
             (CARDS-PLAYED))
```

Re-expressing the search criterion in terms of (CARDS-PLAYED) involves a whole separate derivation over 40 steps long, summarized below and corresponding approximately to steps 6:3-39 of DERIV6.

First the definition of TRICK is expanded:

```
(DURING (TRICK) (TAKE-POINTS ME))

--- [ELABORATE by RULE124] --->

(DURING (SCENARIO
           (EACH P1 (PLAYERS) (PLAY-CARD P1))
           (TAKE-TRICK (TRICK-WINNER)))
        (TAKE-POINTS ME)))
```

This step is performed by

RULE124: $(f\ e_1 \dots e_n) \to e'$, where f is defined as (lambda $(x_1 \dots x_n)$ e) and e' is the result of substituting $e_1 \dots e_n$ for $x_1 \dots x_n$ throughout e

Here f = TRICK.

Next the expression is analyzed to determine which properties of the card played by player ME may cause player ME to take points. Case analysis is used to figure out that player ME can only take points by winning the trick. First the expression is split into two cases:

```
(DURING (SCENARIO
           (EACH P1 (PLAYERS) (PLAY-CARD P1))
           (TAKE-TRICK (TRICK-WINNER)))
        (TAKE-POINTS ME)))
--- [DISTRIBUTE by RULE284] --->
(OR [DURING (EACH P1 (PLAYERS) (PLAY-CARD P1)) (TAKE-POINTS ME)]
    [DURING (TAKE-TRICK (TRICK-WINNER)) (TAKE-POINTS ME)])
```

This split is suggested by

RULE284: $(f \dots (+\ e_1 \dots e_n) \dots) \to (+'\ (f \dots e_1 \dots) \dots (f \dots e_n \dots))$ where + and +' are addition-like operators over the domain and range of f, respectively, and f satisfies the homomorphism identity $f(\dots x + y \dots) = f(\dots x \dots) +' f(\dots y \dots)$

Here f = DURING, + = SCENARIO, and +' = OR.

The first case is eliminated by figuring out that a TAKE-POINTS event can't occur during a sequence consisting exclusively of PLAY-CARD events:

```
(DURING [EACH P1 (PLAYERS) (PLAY-CARD P1)] [TAKE-POINTS ME])
--- [COMPUTE by RULE57] --->
NIL
```

This is accomplished by

RULE57: (during s e) -> nil if s and e have no common sub-events

RULE57's condition is evaluated by an intersection search through the space of defined concepts for a common sub-event of s and e. The sub-events of an event consist of the action concepts in terms of which it's defined. Thus TAKE is a sub-event of

```
TAKE-POINTS = (LAMBDA (P) (FOR-SOME C1 (POINT-CARDS) (TAKE P C1)))
```

Similarly, PLAY is the only sub-event of

```
PLAY-CARD = (LAMBDA (P) (CHOOSE C1 (LEGALCARDS P) (PLAY P C1)))
```

The concepts TAKE-TRICK and TAKE-POINTS have the common sub-event TAKE. Plugging in their definitions and moving some quantifiers around yields:

```
(DURING [TAKE-TRICK (TRICK-WINNER)] [TAKE-POINTS ME])
--- [by analysis] --->
(EXISTS C3 (CARDS-PLAYED)
   (EXISTS C1 (POINT-CARDS)
      (DURING [TAKE (TRICK-WINNER) C3] [TAKE ME C1])))
```

FOO has no definition for the DURING predicate, but knows it's *reflexive*, and can therefore partial-match (TAKE (TRICK-WINNER) C3) against (TAKE ME C1) using

RULE43: (R (f e_1 ... e_n) (f e_1' ... e_n')) -> (and [= e_1 e_1'] ... [= e_n e_n']) if R is reflexive

Here R = DURING and f = TAKE. This produces a quantified conjunction of equalities:

```
(EXISTS C3 (CARDS-PLAYED)
   (EXISTS C1 (POINT-CARDS)
      (DURING [TAKE (TRICK-WINNER) C3] [TAKE ME C1])))
      --- [REDUCE by RULE43] --->
(EXISTS C3 (CARDS-PLAYED)
   (EXISTS C1 (POINT-CARDS)
      (AND [= (TRICK-WINNER) ME] [= C3 C1])))
```

The quantification over C1 is eliminated to produce

```
(AND [EXISTS C3 (CARDS-PLAYED) (HAS-POINTS C3)]
     [= (TRICK-WINNER) ME])
```

The first conjunct in this expression is recognized in terms of the concept HAVE-POINTS:

```
(EXISTS C3 (CARDS-PLAYED) (HAS-POINTS C3))
--- [RECOGNIZE by RULE123] --->
(HAVE-POINTS (CARDS-PLAYED))
```

This recognition is accomplished by

RULE123: e -> (f e_1 ... e_n) if f is defined as (lambda (x_1 ... x_n) <body>)
and e is the result of substituting e_1 ... e_n for x_1 ... x_n throughout <body>

Here f = HAVE-POINTS.

The second conjunct is transformed into a condition on (CARD-OF ME) by analyzing the definition of TRICK-WINNER:

```
(= (TRICK-WINNER) ME)
--- [by analysis] --->
(= (CARD-OF ME) (HIGHEST-IN-SUIT-LED (CARDS-PLAYED)))
```

At this point, the original search criterion (DURING (TRICK) (TAKE-POINTS ME)) has been rephrased in terms of (CARDS-PLAYED):

```
(AND [HAVE-POINTS (CARDS-PLAYED)]
     [= (CARD-OF ME) (HIGHEST-IN-SUIT-LED (CARDS-PLAYED))])
```

The analysis techniques used in the derivation sketched above are described in detail elsewhere (§5). The desired solution-test can now be extracted from the reformulated problem:

```
(REFORMULATE (AND [HAVE-POINTS (CARDS-PLAYED)]
                  [= (CARD-OF ME)
                     (HIGHEST-IN-SUIT-LED (CARDS-PLAYED))])
             (CARDS-PLAYED))

2:19-21 --- [by analysis] --->

([LAMBDA (CARDS-PLAYED1)
   (AND [HAVE-POINTS CARDS-PLAYED1]
        [= (CARDS-PLAYED1 ME)
           (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)])]
 (CARDS-PLAYED))
```

FOO uses this predicate as the solution-test component of the HSM schema, and fills in default values for the step and path constraints:

```
(HSM1 <- SOLUTION-TEST :
      (LAMBDA (CARDS-PLAYED1)
         (AND [HAVE-POINTS CARDS-PLAYED1]
              [= (CARDS-PLAYED1 ME)
                 (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)])))
(HSM1 <- PATH-TEST : T)
(HSM1 <- PATH-ORDER : NIL)
(HSM1 <- STEP-TEST : T)
(HSM1 <- STEP-ORDER : NIL)
(HSM1 <- PATH : CARDS-PLAYED1)
```

This is accomplished by

```
RULE317:    (HSM with    (reformulated-problem : ([lambda (s) Ps] choices)))
      ->    (HSM with    (solution-test : (lambda (s) Ps))
                         (path-test : T) -- default is unpruned search
                         (path-order : nil) -- default is unordered search
                         (step-test : T) -- default is unpruned choice set
                         (step-order : nil) -- default is unordered choice set
                         (path : s)) -- path variable
```

Note that the default values don't represent assumed typical cases (as they do in some knowledge representation schemes), but simply provide initial values that may subsequently be refined by the application of additional knowledge. For example, (PATH-TEST : T) means that no paths are pruned, and (PATH-ORDER : NIL) imposes no ordering on the selection of paths.

3.3.4. Correcting the Initial Formulation of the Search

The HSM schema has so far been instantiated as follows (omitting components used only during the instantiation process itself):

```
(HSM1 WITH
      (CHOICE-SETS : (LAMBDA (P1) (LEGALCARDS P1)))
      (COMPLETION-TEST :
         (LAMBDA (PATH) (= (# PATH) (# (PLAYERS)))))
      (STEP-ORDER : NIL)
      (STEP-TEST : T)
      (PATH-ORDER : NIL)
      (PATH-TEST : T)
      (SOLUTION-TEST :
         (LAMBDA (CARDS-PLAYED1)
               (AND [HAVE-POINTS CARDS-PLAYED1]
                    [= (CARDS-PLAYED1 ME)
                       (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)])))
      (INITIAL-PATH : NIL))
```

This is not quite operational: the generator function (LAMBDA (P1) (LEGALCARDS P1))) is generally uncomputable, since player ME may not look at opponents' cards. (Detecting this problem automatically would require a model that could predict what information would be available to player ME at a given point in the game (§8.1.1).) The problem is solved as follows:

```
                    (LEGALCARDS P1)
2:24                --- [ELABORATE by RULE124] --->
                    (SET-OF C2 (CARDS) (LEGAL P1 C2))
2:26                --- [by RULE318] --->
                    (SET-OF C2 (CARDS) (POSSIBLE (LEGAL P1 C2)))
```

The condition (POSSIBLE P) is true unless P is known to be false (§2.6.3). The revised generator function is computable but too unconstrained: the entire deck of cards will be considered as possible choices for each player unless effective tests can be found with which to reject impossible choices. To find such tests, FOO first expands the definition of LEGAL:

```
    (LEGAL P1 C2)
2:27 --- [ELABORATE by RULE124] --->
    (AND [HAS P1 C2]
         [=> (LEADING P1)
             (OR [CAN-LEAD-HEARTS P1] [NEQ (SUIT-OF C2) H])]
         [=> (FOLLOWING P1)
             (OR [VOID P1 (SUIT-LED)] [IN-SUIT C2 (SUIT-LED)])])
```

Since player `ME` can't inspect opponents' hands, the sub-expression `(HAS P1 C2)` is not directly evaluable. By an interesting process described more fully elsewhere (§2.6.3) (§2.6.3), FOO finds evaluable necessary conditions on `(HAS P1 C2)` and attaches them as annotations, as indicated by the notation "<expression> <- ; <annotation>":

```
2:28-39   --- [by RULE319, analysis] --->
((HAS P1 C2) <- ;
      (=> (OUT C2) IF (IN P1 (OPPONENTS ME))))

2:40-57   --- [by RULE319, analysis] --->
((HAS P1 C2) <- ;
      (=> (NON-VOID P1) IF (IN-SUIT C2 (SUIT-LED))))
```

I assume a runtime evaluation mechanism that can use these annotations when trying to evaluate the annotated expression. For instance, to evaluate `(HAS P1 C2)` for `P1` in `(OPPONENTS ME)`, it would check `(OUT C2)` using the evaluation method derived in DERIV4 (§2.2). A card is `OUT` if some opponent has it. Thus if `(OUT C2)` is false, `P1` doesn't have `C2` and can't play it. This prevents consideration of scenarios in which a card is played twice or an opponent plays a card held by player `ME`. A similar effect would be produced by the annotation

```
((HAS P1 C2) ; (=> (NON-VOID P1) IF (IN-SUIT C2 (SUIT-LED))))
```

This would prevent consideration of scenarios where a player known to be void follows suit.

Such necessary conditions are found by

RULE319: P <- ; (=> Q IF R) if Q's definition directly or indirectly mentions P,
where R is the result of simplifying (=> P Q)

To find a necessary condition for $P(e_1 \dots e_n)$,
Find a predicate Q whose definition mentions P,
And solve for $x_1 \dots x_k$ by reducing the condition $P(e_1 \dots e_n) => Q(x_1 \dots x_k)$.

For example, to find a necessary condition on `(HAS P1 C2)`, FOO searches the knowledge base for predicates defined in terms of `HAS`, and finds

```
OUT = (LAMBDA (C) (EXISTS P (OPPONENTS ME) (HAS P C)))
```

FOO then constructs the expression `(=> (OUT C) (HAS P1 C2))`, which reduces by logical analysis to `(IN P1 (OPPONENTS ME))`, provided `C = C2`. This means that `(OUT C2)` is a necessary condition for `(HAS P1 C2)` whenever `(IN P1 (OPPONENTS ME))` holds.

The generate-and-test procedure represented by the corrected initial instantiation of the HSM schema corresponds to the data-flow graph shown in Figure 3-3.

3.4. Refining HSM by Moving Constraints Between Control Components

In mapping a problem onto HSM, my approach is to start with a "naive" formulation -- *i.e.*, one easy to derive mechanically -- and then refine it step by step into successively better solutions, rather than trying to construct a sophisticated formulation right off the bat. This general approach is illustrated to some extent in DERIV9 (§2.3), where the problem is to decide whether a player is void in a suit. The initial formulation of that problem in terms of the disjoint subsets method only takes into account the number of cards left in the player's hand, but it is then refined to consider in addition the number of cards out in the suit. This is the only refinement performed in DERIV9. In contrast, refinements account for most of the process of formulating a problem as a heuristic search.

In FOO, refinement of HSM consists of transferring a constraint from one control component to another so that it will be applied at an earlier point in the search. The rest of this section illustrates three kinds of refinements corresponding to different kinds of control components to which a constraint can be moved:

1. Applying a test earlier in the search reduces the branching factor of the search by *pruning* branches that can't lead to a solution.
2. The search space can be *reduced* in depth by compiling certain constraints into input parameters and precomputed functions used during the search.
3. *Ordering* the search to consider the most promising paths first helps find a solution path faster when one exists.

3.4.1. Pruning the Search by Applying Tests Earlier

A general strategy for refining a search is to

Reduce the branching factor of the search by pruning partial paths that can't lead to a solution.

Clearly, the key to applying this strategy is to identify partial paths that can't lead to a solution without enumerating all the ways they can be extended into complete sequences.

The solution-test tests whether a complete sequence of choices satisfies the goal of the search. Suppose the solution-test requires that any complete sequence s satisfy some predicate P(s), where if P(s) holds, so must P(s') for every initial subsequence s' of s. Then any path that doesn't satisfy P can safely be pruned from the search, since it can't possibly be extended into a solution path. A predicate P with this property is called a *monotonically necessary condition* on the solution-test.

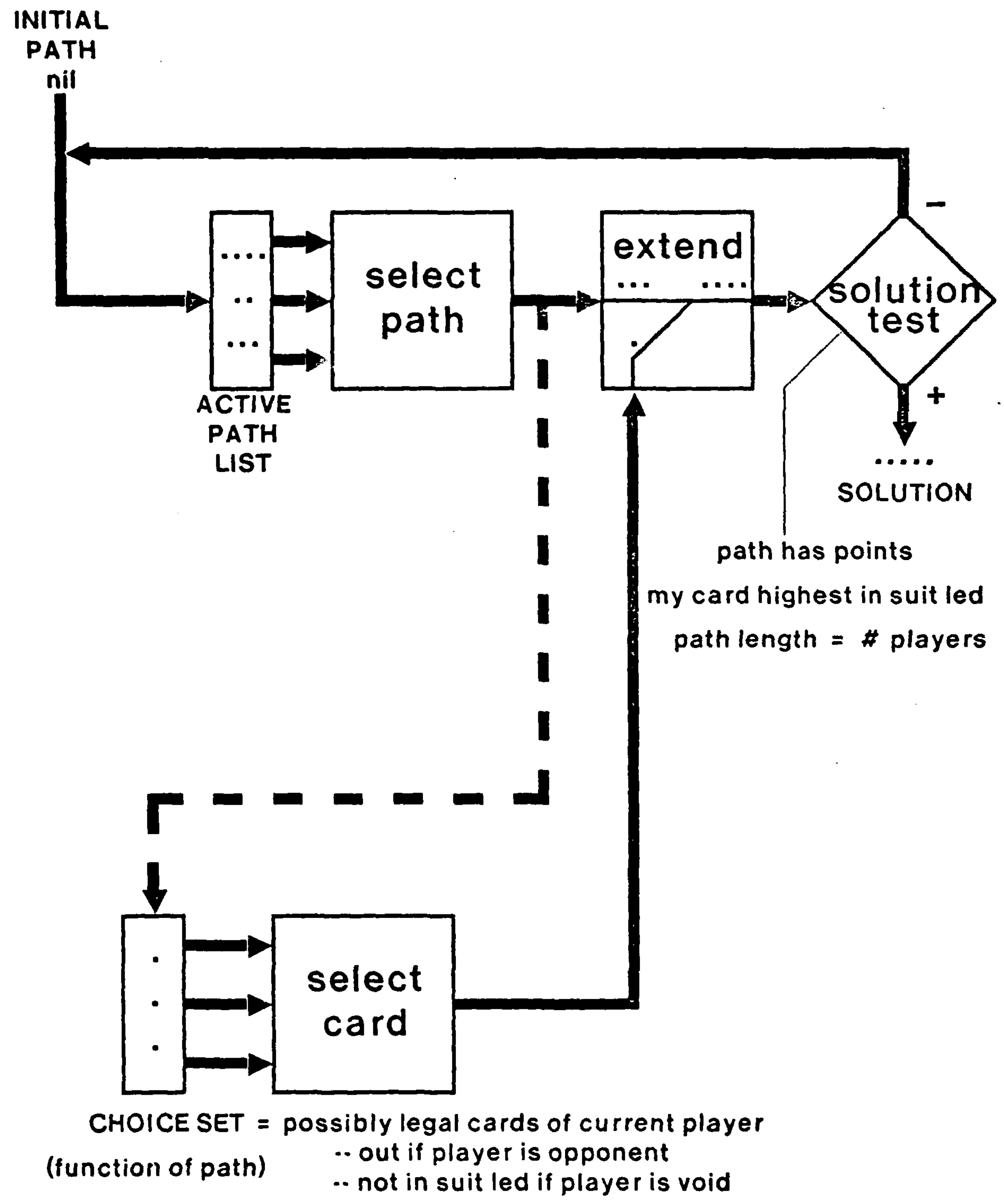
INITIAL PATH nil
ACTIVE PATH LIST
select path
extend
solution test
−
+
SOLUTION
path has points
my card highest in suit led
path length = # players
select card
CHOICE SET = possibly legal cards of current player
(function of path)
-- out if player is opponent
-- not in suit led if player is void

Figure 3-3: Initial Search for Avoid Taking Points

In short, if a necessary condition on the satisfiability of the solution-test (a function of the entire choice sequence) can be formulated purely as a function on a path (initial subsequence of choices), it can be incorporated into the path-test. Similarly, if a necessary condition on the satisfiability of the path-test can be formulated as a test on alternative path extensions, it can be incorporated into the step-test. For example, consider the initial HSM formulation of "avoid taking points," where the path-test is the constant predicate (lambda (s) T), or T for short. [I have relaxed the semantics of lambda calculus to allow an expression e to be treated as the constant function (lambda (x) e). Also, I freely use boolean operators as functionals; thus (and [lambda (x) Px] [lambda (x) Qx]) is interpreted as (lambda (x) (and Px Qx)).] The solution-test is the predicate

```
(LAMBDA (CARDS-PLAYED1)
   (AND [HAVE-POINTS CARDS-PLAYED1]
        [= (CARDS-PLAYED1 ME)
           (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)]))
```

The first conjunct in this expression is not a monotonically necessary condition, since a card sequence that doesn't have points can be extended into one that does simply by appending a point card to it. However, the second conjunct *is* a monotonically necessary condition, since in order for a card sequence CARDS-PLAYED1 to satisfy the conjunct (call it P) every *initial* subsequence of CARDS-PLAYED1 must also satisfy P. That is, in order for player ME's card to be the highest for the whole trick, it must be the *highest at each point in the trick* (once player ME has played). Actually, a slight qualification is required to make P a true monotonically necessary condition. P only applies to paths in which player ME has already played a card, since (CARDS-PLAYED1 ME) is undefined for shorter paths. If the predicate M characterizes those paths in which player ME has already played, then (=> M P) is the desired monotonically necessary condition. This reasoning produces the refinement

```
2:59-73 --- [by RULE323, analysis] --->
(HSM1 <- PATH-TEST :
      (LAMBDA (CARDS-PLAYED1)
         (=> [IN ME (INDICES-OF CARDS-PLAYED1)]
             [= (CARDS-PLAYED1 ME)
                (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)])))
```

The general rule used to make this refinement is

```
RULE323:    (HSM with    (solution-test : (lambda (s) (and ... Ps ...))
                         (path-test : R))
      ->    (HSM with    (path-test : (and R (lambda (s) (=> M Ps)))))
   where M is the result of simplifying the monotonicity condition
   (forall s' (prefixes-of s) (=> Ps Ps'))
```

If the solution-test includes an (almost) monotonically necessary constraint P
Then determine the condition M under which P is monotonically necessary.

And add (=> M P) to the path-test.

In the above example, M is (IN ME (INDICES-OF CARDS-PLAYED1)). Note that RULE323 automatically catches (in M) the sort of exceptions a programmer often forgets. Such details are apparent in the unwieldy but machine-understandable expression produced by moving the highest-card constraint from the path-test to the step-test:

```
2:74-92 --- [by RULE327, analysis] --->
(HSM1 <- STEP-TEST :
      (LAMBDA (P1 C21)
          (=> (NOT (AFTER ME P1))
              (AND [= (SUIT-OF (CARDS-PLAYED1 ME)) (SUIT-LED)]
                   [=> (= (SUIT-OF C21) (SUIT-LED))
                       (NOT (HIGHER C21 (CARDS-PLAYED1 ME)))]))))
```

The effect of this constraint transfer is to prevent the *generation* of paths violating the highest-card constraint, not just prune away already-generated paths. The step-test filters the alternatives C21 at choice point P1 being considered as extensions to the path CARDS-PLAYED1, and rejects cards in the suit led that are higher than player ME's card. In general, this kind of refinement reformulates a constraint on a path as a constraint on a single choice element being considered as a potential path extension. In the latter form, the constraint may be cheaper to test. If only choice elements that satisfy this constraint are used as path extensions, fewer paths will be generated. The reformulated constraint represented by the above step-test can be paraphrased as follows:

If player P1 does not precede player ME,
then only consider C21 as an extension to CARDS-PLAYED1
if player ME's card (in card sequence CARDS-PLAYED1) is in the suit led,
and C21 is not higher than player ME's card if C21 is in the suit led.

The effect of this refinement is to restrict the generation of scenarios to those in which player ME takes the trick. In effect, it reduces the *branching factor* of the search, *i.e.*, the number of alternative path extensions considered at each choice point.

The general rule for moving a constraint from path-test to step-test is

RULE327: (HSM with (path-test : (and ... [lambda (s) Ps] ...))
(step-test : R))
-> (HSM <- (step-test : (and R [lambda (i c) Qc])))
if Ps can be reformulated as (forall i (indices-of s) Qs_i)

If the path-test includes a constraint Ps,
and Ps is equivalent to (forall i (indices-of s) Qs_i) for some predicate Q,
then add the constraint Qc to the step-test.

RULE327 applies when a path-test predicate P on a path $s = s_1 \ldots s_n$ can be recast as a conjunction of the form (and $Qs_1 \ldots Qs_n$) for some predicate Q. In the current example, s is CARDS-PLAYED1 and Ps is

```
(=> [IN ME (INDICES-OF CARDS-PLAYED1)]
    [= (CARDS-PLAYED1 ME) (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)])
```

To apply RULE327, P is first reformulated in the universally quantified form (forall i 1...n Qs_i). Then c = C21 is substituted for s_i = (CARDS-PLAYED1 P1) *in the quantified condition* Qs_i, *and the* resulting expression is simplified to produce Qc =

```
(=> (NOT (AFTER ME P1))
    (AND [= (SUIT-OF (CARDS-PLAYED1 ME)) (SUIT-LED)]
         [=> [= (SUIT-OF C21) (SUIT-LED)]
             [NOT (HIGHER C21 (CARDS-PLAYED1 ME))]]))
```

This is the body of the desired step-test. This *mechanically derived expression is a bit hard to* understand, since it has logical connectives nested four deep, but it can be paraphrased as follows:

If you're considering possible cards for player P1 in a path where player ME follows suit, then don't consider cards in the suit led that are higher than player ME's card.

3.4.2. Reducing the Depth of the Search

Another general strategy for refining a search is to

Reduce the depth of the search by eliminating one or more choice points.

The key to applying this strategy is to use additional problem constraints to compile choices out of the search.

The search procedure so far constructed uses the expression (SUIT-LED) in several places. This is defined as (SUIT-OF (CARD-OF (LEADER))) and therefore depends on the indeterminate quantity (CARDS-PLAYED). To make the procedure operational, this reference must be eliminated. One way would be to replace (SUIT-LED) with

```
(SUIT-OF (CARDS-PLAYED1 (LEADER)))
```

The value of this expression depends on the particular path CARDS-PLAYED1 being considered, and is computed by using (LEADER) as an index into that path. However, there is a better solution if the value of (SUIT-LED) will be known at the time the search starts (and will therefore be the same for all paths considered): simply use that value. This idea is illustrated by a refinement of the step-test:

```
              (SUIT-LED)
    2:94      --- [by RULE333] --->
              SUIT-LED2
    (HSM1 <- VARIABLES : (SUIT-LED2))
    (HSM1 <- BINDINGS : ((SUIT-LED)))
```

Here SUIT-LED2 is a new input variable which will be bound to the value of (SUIT-LED) at the time of the search. The rule for this is

```
RULE333:    (HSM with   (step-test : (... e ...))
                        (sequence : f)
                        (variables : (...))
                        (bindings : (...)))
      ->    (HSM with   (step-test : (... v ...))
                        (variables : (v ...))
                        (bindings : (e ...)))
if e is defined in terms of f.
```

If a sub-expression $P_{f(s)}$ of the step-test refers to the choice sequence, s.
[But the value of f(s) will have been determined by the time the search begins.]
Then change $P_{f(s)}$ to P_v and cache the value of f(s) in v at the beginning of the search.

As RULE333 is currently implemented, the bracketed clause is not tested, since FOO lacks a model of what will be known when. Also, the substitution is performed only within the step-test rather than in all the constraint components, because FOO's rule syntax does not provide a construct for iterating over the components of a schema, although this could easily be corrected. A more general rule would be:

RULE333a: *Cache a choice-sequence-dependent expression whose value won't change during the search.*

Further improvements are possible by exploiting assumptions about when and why the search will be performed. For example, the purpose of the search is to help player ME choose a card, not just to predict whether there's some possible scenario in which ME takes points. Consider the original advice

```
(AVOID (TAKE-POINTS ME) (TRICK)) =

(ACHIEVE (NOT (DURING [TRICK] [TAKE-POINTS ME]))) =

(ACHIEVE (NOT (DURING [SCENARIO (EACH P1 (PLAYERS)
                                   (CHOOSE (CARD-OF P1) (LEGALCARDS P1)
                                      (PLAY P1 (CARD-OF P1))))
                                (TAKE-TRICK (TRICK-WINNER))]
                      [TAKE-POINTS ME])))
```

The search evaluates the condition (DURING ...) by trying to find a value for the non-deterministic card sequence (PROJECT CARD-OF (PLAYERS)) that satisfies the condition. However, since this condition occurs inside the expression (ACHIEVE ...), the implicit purpose of the search is not just to find a scenario in which ME takes points, but to prevent *it from happening*. The condition (DURING [TRICK] [TAKE-POINTS ME]) implicitly depends on the card chosen by the task agent, so avoiding points means *constraining* the choice appropriately (§2.7.1).

This context sensitivity is a weak spot in FOO's knowledge representation, caused partly by the vaguely-defined modal semantics of ACHIEVE and partly by the non-applicative representation of choice (§2.7). It creates a need to detect implicit interactions, since the value of an expression may depend on a variable it does not explicitly mention. Finding such interactions may involve arbitrarily deep expansion of definitions; perhaps it could be done efficiently by means of intersection search through the knowledge base of concept definitions (§5.6). This problem resembles the *aliasing problem* that arises in verifying programs with pointers, where the value of one variable may implicitly depend on the value of another [Luckham 79].

One way to render this implicit interaction explicit is to perform the search separately for each of player ME's cards, and decide whether playing that card might lead to taking points. This involves modifying the search procedure so that it takes player ME's card as an input, and doesn't treat (CARD-OF P1) as a non-deterministic variable when P1 = ME.

Moreover, since the search will not be performed until it's time for player ME to choose a card, the cards played by player ME's predecessors will be known. These ideas are illustrated in refinements on the step-test:

```
                 (CARDS-PLAYED1 ME)
    2:93         --- [by RULE332] --->
                 MY-CARD4
    (HSM1 <- INITIAL-PATH : (PROJECT CARD-OF (PREFIX (PLAYERS) ME)))
    (HSM1 <- BINDINGS : ((CARD-OF ME) (SUIT-LED)))
    (HSM1 <- VARIABLES : (MY-CARD4 SUIT-LED2))
```

Here MY-CARD4 is an input variable which will be bound to a possible value of (CARD-OF ME) before the search begins. Taking (CARD-OF ME) as a given reduces the choice sequence from (EACH P1 (PLAYERS) (PLAY-CARD P1)) to (EACH P1 (OPPONENTS ME) (PLAY-CARD P1)). But since the cards played by player ME's predecessors will be known at search time, the choice sequence can be further reduced to (EACH P1 (PLAYERS-AFTER ME) (PLAY-CARD P1)). This is done by changing the initial-path to (PROJECT CARD-OF (PREFIX (PLAYERS) ME)). Now the search will start with the partial trick scenario consisting of the cards played by everyone up to and including player ME. The effect of this refinement is to reduce the *depth* of the search space, *i.e.*, the *number* of sequential choices required to reach a solution. The general rule for this refinement is

```
RULE332:    (HSM with   (step-test : (... (f' x) ...)))
                        (sequence : f)
                        (choices : f')
                        (indices : S))
       ->   (HSM with   (step-test : (... v ...))
                        (variables : (v ...))
                        (bindings : ((f x) ...))
                        (initial-path : (project f (prefix S x))))
```

Start the search at the point following any choices already determined before search time.

As implemented, RULE332 (like RULE333) revises only the step-test, rather than all the HSM schema components, although this could be rectified straightforwardly. The general idea of RULE332 can be stated in more detail as follows:

> To analyze a property of a choice $f(x_i)$ in a choice sequence $f(x_1) \ldots f(x_n)$,
> use the search space formed by the choice sequence $f(x_{i+1}) \ldots f(x_n)$,
> since at the time $f(x_i)$ is chosen, $f(x_1) \ldots f(x_{i-1})$ will be already be known.

The net effect of the two refinements on the step-test is

```
(HSM1 <- STEP-TEST :
      (LAMBDA (P1 C21)
         (=> [NOT (AFTER ME P1)]
             [AND [= (SUIT-OF MY-CARD4) SUIT-LED2]
                  [=> [= (SUIT-OF C21) SUIT-LED2]
                      [NOT (HIGHER C21 MY-CARD4)]]]])))
```

This step-test is used to filter the generation of path extensions. It works as follows:

> If you're considering whether to extend a path with a card `C21` played by player `P1`,
> Where `P1` plays after player `ME`,
> Make sure that player `ME`'s card is in the suit led,
> And that `C21` isn't both in the suit led and higher than player `ME`'s card.

Given that the search starts after player `ME`'s turn, the proviso `(NOT (AFTER ME P1))` will be true for all paths considered, and therefore superfluous. It would be superfluous even if the search started at the beginning of the trick, thanks to substituting `MY-CARD4` for `(CARDS-PLAYED1 ME)`. That is, if the search is executed separately for each value of `MY-CARD4`, the step-test can be used to avoid considering cases where an opponent plays a card higher than player `ME`'s, *even if the opponent plays before player* `ME`. Generation of the proviso could be eliminated altogether by applying RULE332 to the solution-test *before* moving the highest-card constraint to the path-test and step-test.

The condition `(= (SUIT-OF MY-CARD4) SUIT-LED2)` is independent of `P1`, `C21`, and

CARDS-PLAYED1, and could be tested just once at the beginning of the search, although I didn't implement a rule to make this refinement. This could be done by incorporating an **initial-test** (applied to the **initial-path**) in the generic heuristic search procedure shown in Figure 3-2. This illustrates a limitation of using a fixed set of HSM components: one can always invent refinements that involve adding new components. A more powerful approach would allow refinement rules expressed as transformations on data flow graphs. This is essentially the approach proposed by Tappel [Tappel 80]. An adequacy criterion for a scheme along these lines is whether it can represent the heuristic search procedure and refinement rules presented here.

The last condition restricts the generation of paths to those in which MY-CARD4 is the highest card played in SUIT-LED2:

```
(=> [= (SUIT-OF C21) SUIT-LED2]
    [NOT (HIGHER C21 MY-CARD4)])
```

Note that if player ME leads the trick, SUIT-LED2 must be bound to (SUIT-OF MY-CARD4) before the search begins in order for everything to work. This detail is not addressed in DERIV2. A natural way to deal with it would be to split the whole search procedure into two cases depending on whether player ME leads the trick. Each case would allow refinements appropriate to the assumption on which it was based. However, this would amount to making a separate copy of the flow graph for each case, a transformation that violates the fixed-graph limitation just discussed.

3.4.3. Ordering the Search by Predicting Success

A third strategy for refining a search is to

Order the search to consider first those paths most likely to lead to a solution.

The key to applying this strategy is to identify partial paths likely to lead to a solution without actually trying to extend them into complete sequences.

The refinements described so far have been concerned with pruning branches of the search or eliminating their generation in the first place. The refinements described in this section try instead to *re-order* the search to find a solution faster, assuming one exists. The approach is to find properties of partial solutions that serve as predictors of success, and to give priority to paths that exhibit such properties, which are analytically derived from the search criterion.

The solution-test for the "avoid taking points" example is

```
(LAMBDA (CARDS-PLAYED1)
   (AND [HAVE-POINTS CARDS-PLAYED1]
        [= (CARDS-PLAYED1 ME)
           (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)]))
```

As mentioned before, the second conjunct is a *monotonically necessary condition*, which justifies its incorporation in the path-test and step-test. But the first conjunct is not: although no point cards may have been played so far in a trick, a subsequent player might play one. How then can this constraint be exploited earlier in the search?

The answer lies in the fact that if a path has points, so do all its possible extensions. Thus other things being equal, a path with points is more likely to lead to a solution than one that doesn't, and a solution can be found faster by considering such paths first. (Recall that a "solution" in this search is a card sequence for the trick causing player ME to take points.) In short, the conjunct (HAVE-POINTS CARDS-PLAYED1) is a *monotonically sufficient condition* and can therefore be used as a search-ordering heuristic:

```
2:96-100 --- [by RULE335, analysis] --->
(HSM1 <- PATH-ORDER :
      (LAMBDA (CARDS-PLAYED1)
         (HAVE-POINTS CARDS-PLAYED1)))
```

The general idea behind this refinement is expressed by

RULE335: (HSM with (solution-test : (lambda (s) (and ... Ps ...))))
-> (HSM with (path-order : (lambda (s) Ps)))
if P satisfies the monotonicity condition (forall s' (prefixes-of s) (=> Ps' Ps)).

If the solution-test includes a monotonically sufficient constraint P
Then add P to the path-order.

The have-points constraint can be further exploited by using it to control not only the order in which *existing* paths are considered but also the order in which paths are *generated* to begin with. For example, paths with points can be generated first by using point cards before non-point cards when extending paths in which no points have yet been played. This reasoning is illustrated in:

```
2:101-105 --- [by RULE338, analysis] --->
(HSM1 <- STEP-ORDER :
      (LAMBDA (P1 C21)
         (OR [HAVE-POINTS CARDS-PLAYED1] [HAS-POINTS C21])))
```

The motivation for this step is that a function on a choice element, like HAS-POINTS, may be cheaper to evaluate than a function on a path, like HAVE-POINTS, since it requires less data. That is, testing a single choice element should be faster than testing a sequence of them. The general rule for this refinement is

```
RULE338:    (HSM with    (path-order : (lambda (s) Ps)))
   ->       (HSM with    (step-order : (lambda (i c) Qc)))
where Qc is the result of simplifying (P (append s (list c)))
```

If the path-order contains a constraint P on paths $x_1 \dots x_k$
And $P(x_1 \dots x_k)$ is equivalent to $Q(x_k)$,
Then add Q to the step-order.

An extension to this approach would enhance the data structure representation of a path, s, to include an extra bit, B(s), indicating whether s has points. The value B(s') for a new path s' constructed by appending a card c to s would be computed solely as a function of B(s) and c, not by searching through s' for point cards. This refinement was omitted from DERIV2 because FOO lacks an explicit way to describe data structures (§8.1.5).

3.4.3.1 Refined Formulation of the Search

After the refinements performed by FOO, the HSM schema has been instantiated as follows:

```
(HSM1 WITH

      (VARIABLES : (SUIT-LED2 MY-CARD4))
      (BINDINGS : ((SUIT-LED) (CARD-OF ME)))
      (INITIAL-PATH : (PROJECT CARD-OF (PREFIX (PLAYERS) ME))))

      (CHOICE-SETS :
         (LAMBDA (P1)
            (SET-OF C2 (CARDS)
               (POSSIBLE
                  (AND [HAS P1 C2]
                       : (=> (OUT C2)
                          IF (IN P1 (OPPONENTS ME)))
                       ; (=> (NON-VOID P1)
                          IF (IN-SUIT C2 (SUIT-LED)))
                       [=> (LEADING P1)
                           (OR [CAN-LEAD-HEARTS P1]
                               [NEQ (SUIT-OF C2) H])]
                       [=> (FOLLOWING P1)
                           (OR [VOID P1 (SUIT-LED)]
                               [IN-SUIT C2 (SUIT-LED)])]))))))

      (STEP-ORDER :
         (LAMBDA (P1 C21)
            (OR [HAVE-POINTS CARDS-PLAYED1] [HAS-POINTS C21])))
      (STEP-TEST :
         (LAMBDA (P1 C21)
            (=> [NOT (AFTER ME P1)]
                [AND [= (SUIT-OF MY-CARD4) SUIT-LED2]
                     [=> [= (SUIT-OF C21) SUIT-LED2]
                         [NOT (HIGHER C21 MY-CARD4)]]]))))
```

```
(PATH-ORDER :
   (LAMBDA (CARDS-PLAYED1) (HAVE-POINTS CARDS-PLAYED1)))
(PATH-TEST :
   (LAMBDA (CARDS-PLAYED1)
       (=> [IN ME (INDICES-OF CARDS-PLAYED1)]
           [= (CARDS-PLAYED1 ME)
              (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)])))

(COMPLETION-TEST :
   (LAMBDA (PATH) (= (# PATH) (# (PLAYERS)))))
(SOLUTION-TEST :
   (LAMBDA (CARDS-PLAYED1)
      (AND [HAVE-POINTS CARDS-PLAYED1]
           [= (CARDS-PLAYED1 ME)
              (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)])))
```

The search thereby specified is represented as a data-flow graph in Figure 3-4, and has the following features.

The object of the search is to find a sequence of cards, CARDS-PLAYED1, indexed by the sequence of players, that satisfies the solution-test and completion-test. The solution-test requires that the sequence contain one or more point cards and that the highest card in the suit led be player ME's card. The completion-test requires that the sequence contain one card for each player. The combined effect is to specify a sequence of cards that will cause player ME to take points.

The choice set at each point in the search, indexed by player variable P1, consists of those cards that might possibly be legal for P1 to play, *i.e.*, whose legality for P1 is not contradicted by information available to player ME. For example, in order for a card C2 to be legal for P1, P1 must have C2. If the truth value of (HAS P1 C2) doesn't happen to be known at search time, player ME can check a couple of necessary conditions. The first is that C2 be out if P1 is an opponent of ME; in particular, C2 cannot already have been played. This prevents consideration of scenarios in which the same card is played more than once, or in which a card previously taken miraculously reappears. The second necessary condition is that P1 be non-void in the suit led if C2 is in the suit led. A more general test would check that P1 is not known to be void in (SUIT-OF C2).

The search conservatively takes as the choice set all cards satisfying the known necessary conditions; thus an effective way to reduce the branching factor of the search would be to derive additional necessary conditions. For example, a third necessary condition would be (VOID P1) when (IN-SUIT C2 (SUIT-LED)); this would be of little help, however, without some non-trivial necessary conditions on VOID. Obvious methods for evaluating VOID typically give sufficient rather than necessary conditions; it's hard to be sure that an opponent is *non*-void.

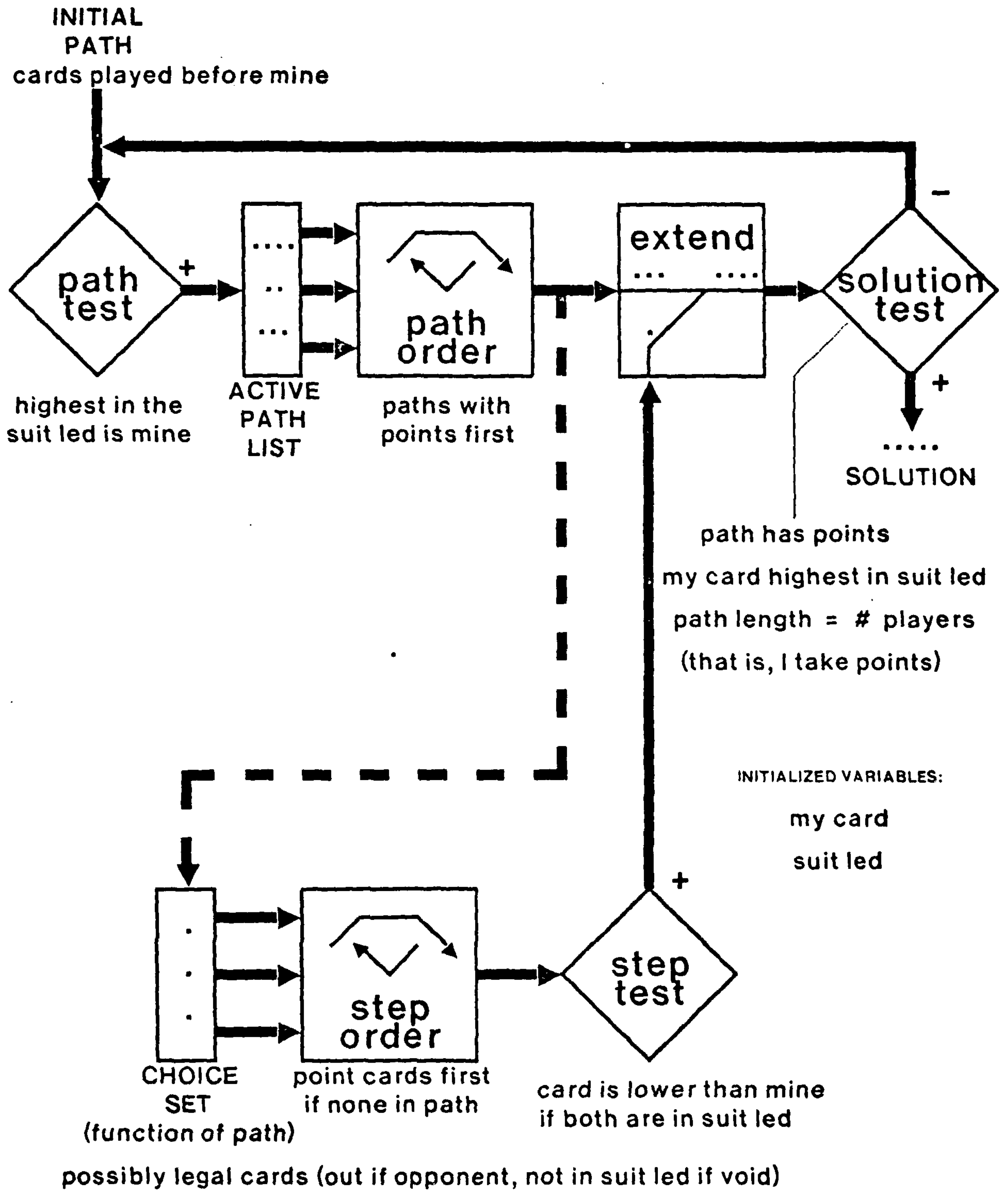

Figure 3-4: Refined Search for Avoid Taking Points

The search uses lambda variables SUIT-LED2 and MY-CARD4, which can be bound to values computed or selected at the outset of the search.

The search proceeds by choosing a path from the search space, extending it by one element, testing the result to see if it's a solution, and deciding whether to add it to the search space. Initially the only path in the search space consists of the cards played by player ME's predecessors, plus MY-CARD4, the card being considered for (CARD-OF ME).

The path-order is used to constrain the order in which paths are selected for consideration, so that paths in which points have been played are considered first.

The step-test is used to test possible extensions, so that paths in which player ME doesn't win the trick are not even considered.

The step-order is used to order possible extensions, so that point cards are considered first when extending a path in which no points have yet been played.

The path-test prunes paths in which player ME can't take the trick no matter what cards are played in the rest of the trick. Since the step-test prevents the generation of such paths, the path-test need not really be applied to *any paths other than the initial-path.* RULE323 could be fixed to reflect this by adding an initial-test component to the HSM schema, but the major benefit of moving constraints earlier in the search comes from combinatorially reducing the size of the search; thus the constant-factor cost of a redundant test should be negligible compared to the unconstrained search executed in the absence of a step-test.

I have not implemented the search process itself beyond describing it, but it seems straightforward enough. It would require some clean-up, *e.g.*, rewriting (CARD-OF ME) and (CARDS-PLAYED1 ME) as MY-CARD4 throughout the instantiated schema. The need for such global substitutions comes from waiting until the method is already half-instantiated before applying high-level transformations, *e.g.*, deciding to start the search after (CARD-OF ME) instead of at the beginning of the trick. I wriggled out of this by adding an initial-path component, but in general one would expect such major decisions to require redoing a lot of the work done up to that point, or at least fixing it up.

Finally, the instantiated search is designed to be executed in the environment used for evaluating expressions. This environment provides things like 3-valued logic (unevaluable expressions return a value UNKNOWN rather than causing runtime errors), a state model (in which actions can be simulated), historical reference (*e.g.*, the WAS-DURING construct), and multiple evaluation methods (annotations). Such features seem reasonably straightforward to build into an evaluator (if one doesn't care about efficiency), but it would take a fair amount of work and I haven't done it. Instead, I've satisfied myself with giving specifications for such an evaluator and showing how particular features in the evaluator (*e.g.*, annotations) can simplify the solution of particular kinds of operationalization problems (*e.g.*, sometimes-evaluable expressions). None of these features is particularly original so far as I know; for instance, Balzer has described a compilable formal specification language that permits historical reference [Balzer 79].

3.4.4. Moving to a Smaller Search Space

(§3.4.1) described a way to reduce the effective branching factor of the search by using a step-test to filter the generation of alternatives at each choice point. (In contrast, reducing the number of choice points reduces the *depth* of the search space.) A more sophisticated refinement method, which I have not implemented, would reduce the branching factor by moving the search to an abstracted version of the original search space:

> To reduce a search space, suppress operator details that are irrelevant to the search criterion, and use the resulting abstractions in place of the original operators [Mostow 79a].

Abstractions of operators have been constructed and used to reduce search in planning systems, but the abstraction process has been limited: operators have been abstracted by deleting some of their preconditions [Newell 60] [Sacerdoti 77] or by ignoring variable bindings [Sacerdoti 74] [Klahr 78]. In contrast, the abstractions proposed below would be synthesized based on analysis of the search criterion and knowledge about the task domain.

For instance, from the point of view of avoiding taking points, it often doesn't matter what suit an opponent plays (just whether it's in the suit led or has points), or exactly how low an opponent's card is (as long as it's lower than (CARD-OF ME)). This suggests reducing the generator set from a set of cards to a set of abstract alternatives like {UNDER-PLAY, DROP-POINTS, BREAK-SUIT}. These abstract alternatives represent equivalence classes of concrete ones, and the equivalence relation strongly depends on the task situation. For instance, in some cases playing any heart will have the same impact on the outcome of the trick, while in others the particular heart played might determine who wins the trick. An obvious approach to this problem is to look dynamically for "easy" equivalences and use them to reduce the search space for the particular situation at hand according to the heuristic

> *Ignore an alternative if an equivalent one has already been considered.*

Suppose an abstract alternative like UNDER-PLAY is defined as the set of all choices x satisfying P(x). Then

1. Look for sufficient conditions S(x, x') for (P(x) <=> P(x')).
2. Partition the generator set into equivalence classes Cx = {x' | S(x, x')}.
3. Replace the old generator set with a new one containing a single representative element of each class. Since the revised generator produces the same kind of elements as the old one (rather than some kind of abstracted element), the other control components still work.
4. Perform the search as usual.

A similar approach is to

Ignore an alternative if a better one has already been considered.

This approach is based on the concept of *extreme cases* [Lenat 78]:

1. Find a sufficient condition Better(s, s') ("s dominates s'") such that if s' can be extended into a solution then so can s. The predicate Better(s, s') can be used to order paths by considering s before s' if s dominates s'.
2. Use a path-test Best(s) = ~∃s' | Better(s', s) reflecting the fact that if an "easiest case" s fails, so will every path s' known to be harder than it.
3. Use a step-test Best(s & c), where s represents the sequence of choices made before c.

A disadvantage of this approach lies in having to repeat the dynamic search for sufficient conditions in each new situation. Nonetheless, this might capture an aspect of human play -- figuring out what will happen if player p plays card x, and then jumping to the conclusion that the same thing will happen if p plays any other card in the class Cx. The major obstacle in implementing this approach, of course, would be to mechanize the analysis involved in identifying evaluable sufficient conditions for equivalence or domination.

3.4.5. Refining HSM: Summary

FOO's HSM refinement rules transfer constraints from one component of the HSM schema to another, as shown in Figure 3-5. Some rules prune search paths that can't lead to a solution. Other rules re-order the search to consider the most promising paths first. F. ally, some rules compile constraints out of the search altogether. In general, the rules transfer constraints so as to apply them earlier in the search, as shown in Figure 3-6.

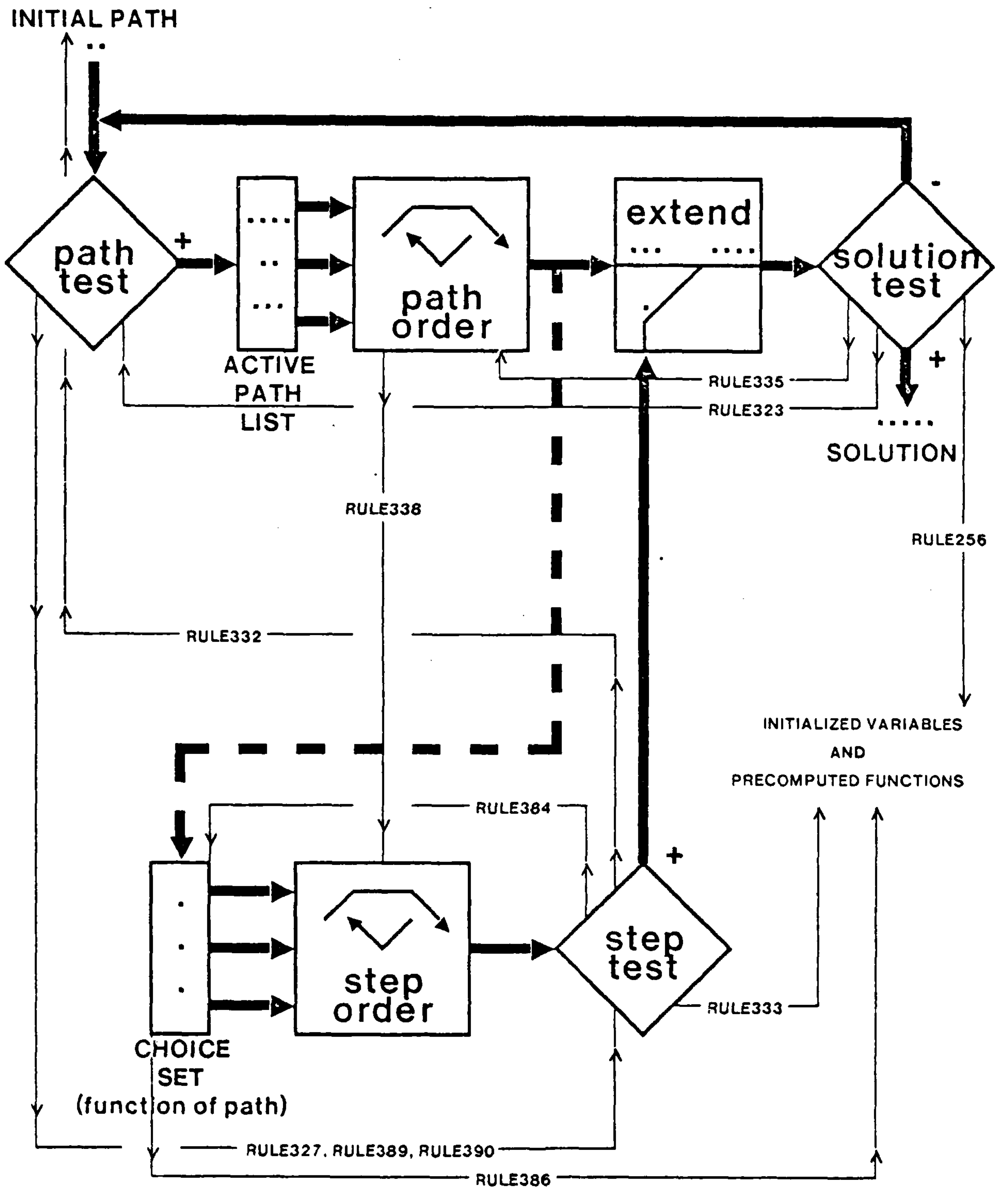

Figure 3-5: HSM Procedure and Refinement Rules

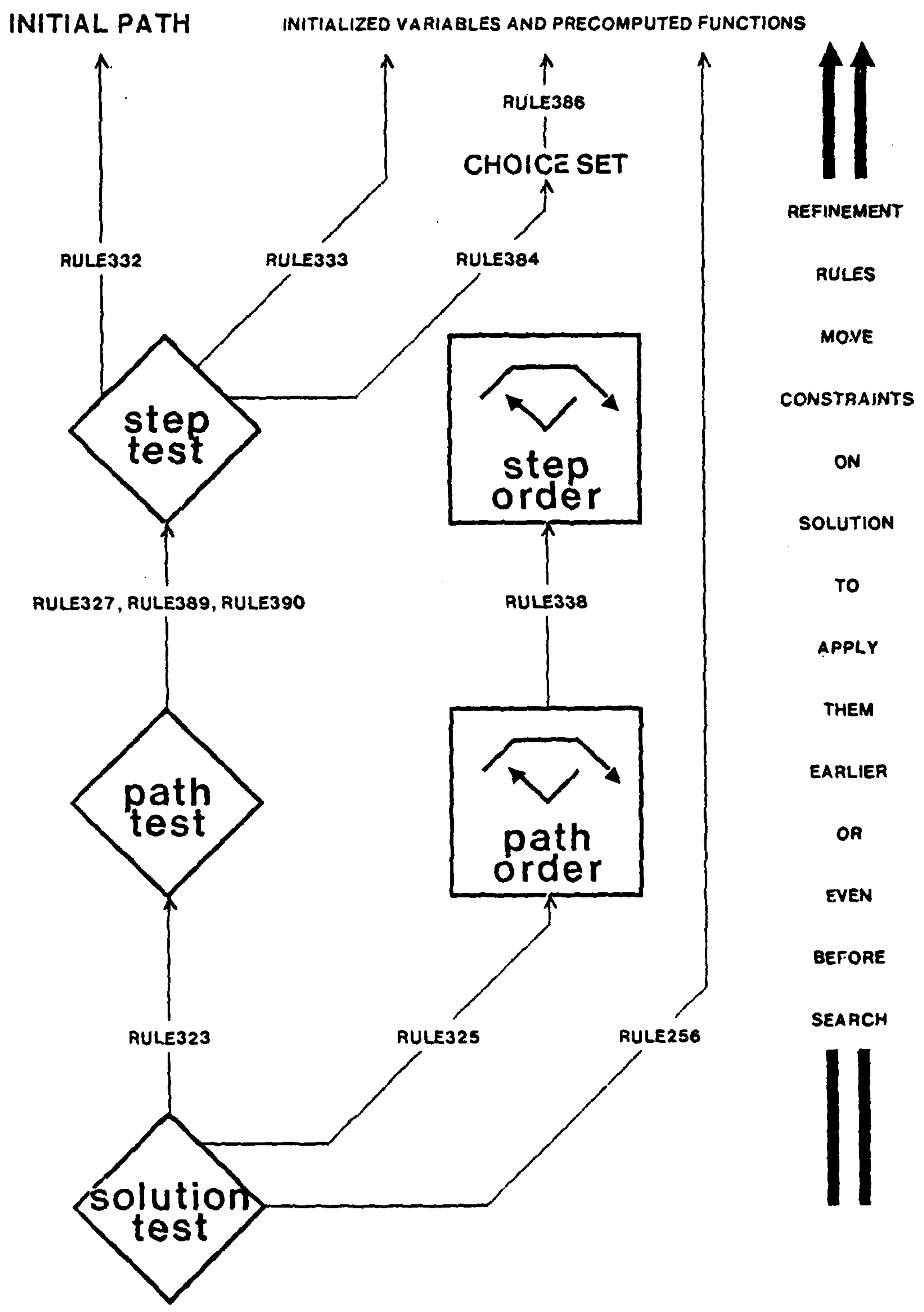

Figure 3-6: HSM Refinement Rules

3.5. Another Example: Compose a Cantus

So far FOO's HSM schema and its rules for instantiating and refining this schema have been illustrated solely in terms of the "avoid taking points" example. As a test of generality, I applied the same rules to a task from the domain of music: "compose a cantus firmus." A *cantus firmus* is a sequence of musical tones of equal length satisfying certain aesthetic constraints, and forms the melodic backbone of a piece of music.

A program to generate such sequences was implemented by Meehan [Meehan 72] using aesthetic constraints given in a textbook on counterpoint [Salzer 69]. Meehan structured his program as a depth-first search through the space of tone sequences, and used the twenty or so textbook constraints to prune the search. Meehan has said that this approach "isn't how music is or should be written, any more than a recursive enumeration of all the legal strings of some nice grammar would describe the generation of natural language;"[7] in [Meehan 79], he proposes an AI approach to music composition. Although this constraint satisfaction problem may be considered irrelevant to real music, it provides a well-defined symbolic task suitable as a vehicle for evaluating the generality of FOO's knowledge about HSM.

For my experiment, I took four constraints from [Salzer 69]:

C1. "As a rule the cantus will not contain fewer than eight or more than sixteen tones."

C2. "The cantus firmus should not contain intervals larger than an octave, dissonant leaps, or chromatic half steps."

C3. "A tenth between the lowest and the highest tone is the maximum range."

C4. "Each cantus firmus must contain a climax or high point.... The climax tone should not be repeated."

I encoded these constraints in FOO as follows:

```
C1:  (IN (LENGTH (CANTUS)) (LB:UB 8 16))
C2:  (FORALL X (INTERVALS-OF (CANTUS)) (USABLE-INTERVAL X))
C3:  (=< (MELODIC-RANGE (CANTUS)) (MAJOR 10))
C4:  (= (#OCCURRENCES (CLIMAX (CANTUS)) (CANTUS)) 1)
```

The goal of the experiment was to operationalize these constraints mechanically in terms of HSM

[7]Meehan, personal communication.

using the same rules used in the "avoid taking points" example, or similar ones. The task of composing a cantus firmus was represented as

```
(ACHIEVE (LEGAL-CANTUS (CANTUS)))
```

Here the predicate LEGAL-CANTUS tests whether a given tone sequence satisfies C1-C4, and CANTUS denotes the act of composing a cantus by choosing each note in succession:

```
(CANTUS) = (EACH I (LB:UB 1 (CANTUS-LENGTH)) (CHOOSE-NOTE I))
```

The term (CANTUS-LENGTH) is undefined; it appears because FOO lacks a construct for defining a cantus as a *sequence of unspecified length* -- a beast more type-like than object-like, in that it describes a class of possible objects. Such an entity could be defined intensionally by a predicate like

```
(IS-CANTUS S) = (FORALL I (LB:UB 1 (LENGTH S))
                    (IS-CHOOSE-NOTE I (S I)))
```

"A cantus is a sequence of events the i^{th} of which consists of choosing the i^{th} note."

However, I wanted (CANTUS) to denote the cantus itself, not a truth value. DERIV14 circumvents this problem by using the variable CANTUS to denote the cantus, with the implicit assumption that CANTUS satisfies the predicate IS-CANTUS.

The rest of this section parallels the description of the "avoid taking points" example in (§3.3) through (§3.4.3.1). (§3.5.1) describes the process of instantiating the HSM schema for the composition task. (§3.5.2) presents the initial instantiation. (§3.5.3) describes a series of subsequent refinements to the search and the general rules used to make them. (§3.5.4) describes the refined search. Both the initial formulation and the subsequent refinement of the search involve several HSM rules used in the Hearts example as well. (§3.5.5) presents further evidence for the generality of these rules by showing how they could be used to make additional refinements in the music example.

3.5.1. Formulating the Search

The first step in mapping the problem (ACHIEVE (LEGAL-CANTUS (CANTUS))) onto the HSM schema is to recognize the relevance of HSM to the problem. This is effected by the same rule that recognizes "avoid taking points" as a heuristic search problem:

RULE306: (P ... e ...)
-> (HSM with (problem : (P ... e ...)) -- *expression to be evaluated*
(object : e) -- *scenario containing choices*
(choice-seq : (choice-seq-of e)) -- *sequence of choices made in scenario*
if e contains an event sequence involving choice

Use heuristic search to evaluate a predicate on a scenario in which choices are made.

Thus DERIV13, the derivation for the "compose a cantus firmus" example, begins as follows:

```
        (LEGAL-CANTUS (CANTUS))
13:1    --- [by RULE306] --->
        HSM1
(HSM1 <- PROBLEM : (LEGAL-CANTUS (CANTUS)))
(HSM1 <- OBJECT : (CANTUS))
(HSM1 <- CHOICE-SEQ : (CHOICE-SEQ-OF (CANTUS)))
```

In this case the predicate P is LEGAL-CANTUS and the object e is the event sequence

```
(CANTUS) = (EACH I (LB:UB 1 (CANTUS-LENGTH)) (CHOOSE-NOTE I))
```

This scenario involves the choice event

```
(CHOOSE-NOTE (NOTE I) (TONES) (NOTE I))
```

As before, the next step is to identify the sequence of choices involved. The same rules are used. (Unless otherwise noted, the HSM rules used below are the same ones developed for the "avoid taking points" example, although the analysis required is different.)

```
          (CHOICE-SEQ-OF (CANTUS))
13:2-8    --- [by RULE307-309, analysis] --->
          (EACH I1 (LB:UB 1 (CANTUS-LENGTH))
             (CHOOSE (NOTE I1) (TONES) (NOTE I1)))
```

This enables several components of the HSM schema to be instantiated:

```
13:9     --- [ELABORATE by RULE311] --->
(HSM1 <- SEQUENCE : NOTE)
(HSM1 <- CHOICES : (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))))
(HSM1 <- CHOICE-SETS : (LAMBDA (I1) (TONES)))
(HSM1 <- INDICES : (LB:UB 1 (CANTUS-LENGTH)))
(HSM1 <- INDEX : I1)
(HSM1 <- VARIABLES : NIL)
(HSM1 <- BINDINGS : NIL)
(HSM1 <- INITIAL-PATH : NIL)
(HSM1 <- COMPLETION-TEST :
      (LAMBDA (PATH)
         (= (LENGTH PATH) (LENGTH (LB:UB 1 (CANTUS-LENGTH))))))
```

For example, the choice set for the I1th note in the cantus is specified to be the entire set of

(TONES) available, and is thus independent of I1. (However, a subsequent refinement narrows the choice set at each point in a way that depends on the previous note.)

The search criterion can now be reformulated in terms of the identified choice sequence. The the solution-test is extracted from the resulting expression, and some defaults are filled in:

```
(REFORMULATE (LEGAL-CANTUS (CANTUS))
             (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))))
13:10-18 --- [by RULE315, RULE317, analysis] --->
(HSM1 <- SOLUTION-TEST :
       (LAMBDA (PROJECT1)
          (AND
C1:              [IN (LENGTH PROJECT1) (LB:UB 8 16)]
C2:              [FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                    (USABLE-INTERVAL INTERVAL1)]
C3:              [=< (MELODIC-RANGE PROJECT1) (MAJOR 10)]
C4:              [= (#OCCURRENCES (CLIMAX PROJECT1) PROJECT1) 1])))
(HSM1 <- PATH-TEST : T)
(HSM1 <- PATH-ORDER : NIL)
(HSM1 <- STEP-TEST : T)
(HSM1 <- STEP-ORDER : NIL)
(HSM1 <- PATH : PROJECT1)
```

So far instantiation of the HSM schema for the cantus example has paralleled the initial phase of the "avoid taking points" example. At this point it diverges in order to fix up the completion-test

```
(LAMBDA (PATH)
   (= (LENGTH PATH) (LENGTH (LB:UB 1 (CANTUS-LENGTH)))))
```

This test is non-operational since CANTUS-LENGTH is undefined. Fortunately constraint C1 gives a criterion for identifying complete paths, so it is moved from the solution-test to the completion-test:

```
13:19    --- [by RULE378] --->

(HSM1 <- COMPLETION-TEST :
       (LAMBDA (PROJECT1) (IN (LENGTH PROJECT1) (LB:UB 8 16))))

(HSM1 <- SOLUTION-TEST :
       (LAMBDA (PROJECT1)
          (AND [FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                  (USABLE-INTERVAL INTERVAL1)]
               [=< (MELODIC-RANGE PROJECT1) (MAJOR 10)]
               [= (#OCCURRENCES (CLIMAX PROJECT1) PROJECT1) 1])))
```

This correction is made by a rule not used in the "avoid taking points" example:

RULE378: (HSM with (solution-test : (lambda (s) (and ... (P (length s)) ...))))
-> (HSM with (completion-test : (lambda (s) (P (length s))))
(solution-test : (lambda (s) (and ...))))
assuming the previous completion-test was undefined or subsumed by the new one

If the solution-test contains a constraint on path length, move it to the completion-test.

The idea of this rule is that a test that depends solely on the length of a path is a halting criterion and therefore belongs in the completion-test. Actually, this step would be unnecessary if the representation allowed (CANTUS-LENGTH) to be defined as "any non-negative integer", so that the original completion-test would reduce to T:

```
(LAMBDA (PATH)
   (= (LENGTH PATH) (LENGTH (LB:UB 1 (CANTUS-LENGTH)))))

--- [by definition of LB:UB] --->
(LAMBDA (PATH) (= (LENGTH PATH) (CANTUS-LENGTH)))

--- [by definition of CANTUS-LENGTH] --->
(LAMBDA (PATH) (= (LENGTH PATH) <non-negative integer>))

--- [by definition of <non-negative integer>] --->
(LAMBDA (PATH) (>= (LENGTH PATH) 0))

--- [by definition of LENGTH] --->
(LAMBDA (PATH) T)
```

The problem with this is the implicit ambiguous quantification in the expression

```
(= (LENGTH PATH) <non-negative integer>)
```

The intended meaning of this expression is

```
(EXISTS I (NON-NEGATIVE-INTEGERS) (= (LENGTH PATH) I))
```

However, the syntax of the first expression fails to specify the scope of the quantifier EXISTS. For example, if (= (LENGTH PATH) <non-negative integer>) occurred as part of a larger boolean expression, would the quantifier apply to the sub-expression or to the whole thing? The quantification issue in knowledge representation has been discussed elsewhere, e.g., in [Woods 75].

3.5.2. Initial Formulation of the Search

The components of the HSM schema have now been instantiated as follows:

```
    (HSM1 WITH
        (INITIAL-PATH : NIL))
        (CHOICE-SETS : (LAMBDA (I1) (TONES))))
        (STEP-ORDER : NIL)
        (STEP-TEST : T)
        (PATH-ORDER : NIL)
        (PATH-TEST : T)
        (COMPLETION-TEST :
    C1:    (LAMBDA (PROJECT1) (IN (LENGTH PROJECT1) (LB:UB 8 16))))
        (SOLUTION-TEST :
           (LAMBDA (PROJECT1)
    C2:       (AND [FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                      (USABLE-INTERVAL INTERVAL1)]
    C3:            [=< (MELODIC-RANGE PROJECT1) (MAJOR 10)]
    C4:            [= (#OCCURRENCES (CLIMAX PROJECT1) PROJECT1) 1]))))
```

The search thereby specified corresponds to the generate-and-test procedure shown in Figure 3-7. Although executable in theory, it would prove combinatorially inefficient in practice, since the constraints C1-C4 are not used to constrain the search.

3.5.3. Refining the Search

The refinement rules introduced in the "avoid taking points" example, and others like them, can now be used to move constraints earlier in the search. This section discusses the constraints C2, C3, and C4 in turn. (Recall that C1 has already been moved from solution-test to completion-test.)

3.5.3.1 Moving C2 to the step-test

The constraint C2 appears in the solution-test as the clause

```
(FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
   (USABLE-INTERVAL INTERVAL1))
```

It is moved to the path-test by

RULE323: *If the solution-test includes an (almost) monotonically necessary constraint P*
Then determine the condition M under which P is monotonically necessary,
And add (=> M P) to the path-test.

Here the monotonicity condition M reduces to T, yielding

```
13:20-27 --- [by RULE323, analysis] --->
(HSM1 <- PATH-TEST :
      (LAMBDA (PROJECT1)
         (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
            (USABLE-INTERVAL INTERVAL1))))
```

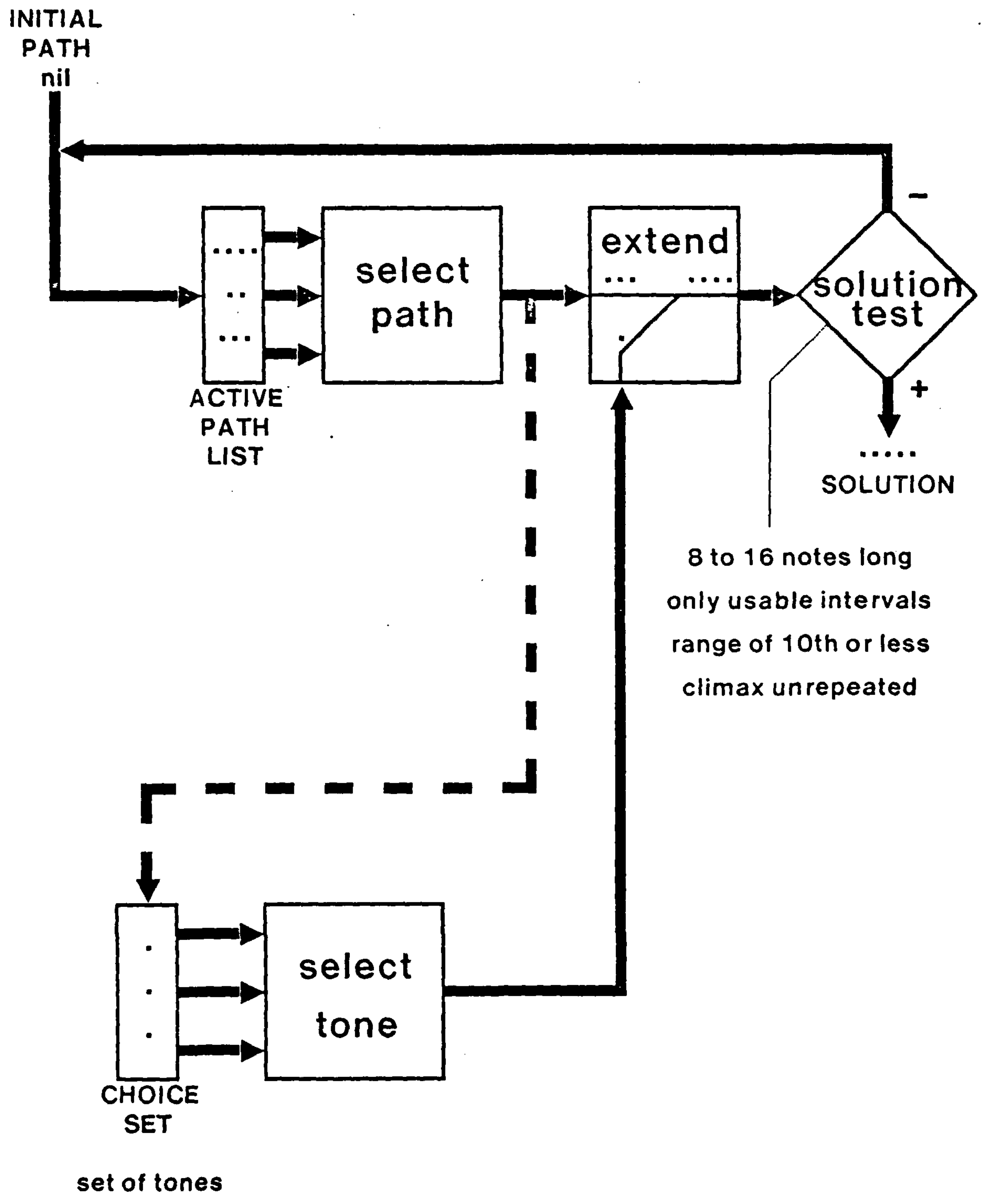

Figure 3-7: Initial Search for Compose Cantus

Constraint C2 is then moved to the step-test by

> RULE327: *If the path-test includes a constraint Ps,*
> *And Ps is equivalent to (forall i (indices-of s) Qs) for some predicate Q,*
> *Then add the constraint Qc to the step-test.*

This involves a change of variable so as to quantify over notes instead of intervals. The requantified constraint specifies that each note is a usable interval away from its predecessor:

```
13:28-43 --- [by RULE327, analysis] --->
(HSM1 <- STEP-TEST :
      (LAMBDA (I1 NOTE1)
       (OR [USABLE-INTERVAL (INTERVAL (PROJECT1 (- I1 1)) NOTE1)]
           [= I1 1])))
```

The exception for I1 = 1 indicates that this constraint does not apply to the generation of the first note of the cantus, since that note has no predecessors with which to form an interval. The effect of moving C2 to the step-test is to prevent the extension of paths by intervals that fail to satisfy the USABLE-INTERVAL predicate.

3.5.3.2 Precomputing C2

A further improvement is achieved by constraining the next-note generator to satisfy C2 to begin with. First C2 is moved into the definition of the choice set:

```
13:52 --- [REFINE by RULE384] --->
(HSM1 <- CHOICE-SETS :
            (LAMBDA (I1)
               (SET-OF NOTE1 (TONES)
                  (OR [USABLE-INTERVAL
                          (INTERVAL (PROJECT1 (- I1 1)) NOTE1)]
                      [= I1 1]))))
(HSM1 <- STEP-TEST : T)
```

This is accomplished by a new rule (*i.e.*, one not used in the Hearts example):

```
RULE384:     (HSM with     (step-test : (lambda (i c)  Pic))
                           (choice-sets : (lambda (i) Si)))
      ->     (HSM with     (step-test : T)
                           (choice-sets : (lambda (i) (set-of c Si Pic))))
```

> *If the step-test includes a constraint P,*
> *Change the choice set S to be {x in S | Px}.*

This transformation by itself doesn't improve the efficiency of the search, since it merely moves the generate-and-test cycle into the evaluation of CHOICE-SETS. In this instance, however, the test can be moved out of the search altogether by *precomputing* the function

```
USABLE-SUCCESSORS-OF =
  (LAMBDA (NOTE1)
     (SET-OF NOTE2 (TONES)
        (USABLE-INTERVAL (INTERVAL NOTE1 NOTE2))))
```

The function USABLE-SUCCESSORS-OF is precomputable since it depends only on the last note, NOTE1, of the path PROJECT1. If the set (TONES) is reasonably small, it is feasible to precompute and store in a table the usable successors of each tone. The generator can then be revised to retrieve the usable successors of the last note:

```
13:53-56 --- [by RULE386, analysis] --->
(HSM1 <- CHOICE-SETS :
      (LAMBDA (I1)
         (IF (= I1 1)
             (TONES)
             (USABLE-SUCCESSORS-OF (PROJECT1 (- I1 1))))))
```

The function USABLE-SUCCESSORS-OF was synthesized by FOO. (I provided a name for the function when asked by FOO to do so.) The general rule for precomputing the choice set is

```
RULE386:     (HSM with      (path : s)
                            (choice-sets : (... (set-of x S (P ... (g s) ...)) ...)))
      ->     (HSM with (choice-sets : (f (g s))))
f <- (lambda (y) (set-of x S (P ... y ...)))
if (P ...) is otherwise independent of s
```

If the choice set has the form {x in S | P(x, g(s))}
[Where S and range(g) are small
And P doesn't otherwise depend on the path variable s.]
Then define a new function f(y) = {x in S | P(x, y)}
And change the choice set to f(g(s)),
Where f is precomputed and stored in a table before the search begins.

As implemented, FOO does not test the bracketed clauses of RULE386. Here S = (TONES) and g = (PROJECT1 (- I1 1)). The size of (TONES) could be determined by counting it, given an extensional definition, or by asking the user. The fact that the range of PROJECT1 (considered as a function from integers to notes) is a subset of (TONES) could be determined by analysis. The space cost of storing f in a table, say as an array of lists of tones, is easy to predict given the average frequency with which the predicate USABLE-INTERVAL! is satisfied. Computations of a similar sort have been performed by systems that evaluate alternative data structures for a given application [Kant 79] [Low 78].

A potentially difficult aspect of RULE386 is to verify that P does not depend implicitly on s. An intersection search could be used to determine whether any of the concepts in P mentioned s in their definitions, but the approach would have to be refined to account for an aliasing problem: the sequence might be referred to by a name other than PROJECT1, *e.g.*, (CANTUS) or (NOTE I1).

Note that constraint C2, originally part of the solution-test, has now been factored out of the search altogether. Instead of considering the entire set of (TONES) when extending a path (tone sequence), the search will only consider the legal successor tones of the last note. The effect of this refinement is to reduce the branching factor of the search.

3.5.3.3 Moving C3 to the Path-test

Constraint C3 appears in the solution-test as

```
(=< (MELODIC-RANGE PROJECT1) (MAJOR 10))
```

C3 is moved from the solution-test to the path-test by the same rule used previously:

```
13:44-51  --- [by RULE323, analysis] --->
(HSM1 <- PATH-TEST :
      (AND ... [LAMBDA (PROJECT1)
                   (=< (MELODIC-RANGE PROJECT1) (MAJOR 10))]))
```

The effect of this refinement is to prune paths as soon as their range exceeds the allowable limit of a major tenth, thereby preventing the consideration of their various extensions.

In principle, RULE327 could be used to reformulate the path-test version of C3 as the step-test

```
(AND [=< (SIZE (INTERVAL (LOWEST PROJECT1) NOTE1)) (MAJOR 10)]
     [=< (SIZE (INTERVAL NOTE1 (HIGHEST PROJECT1)) (MAJOR 10)])
```

However, this would require a considerable amount of analysis based on the mathematical properties of intervals, and I didn't do it.

3.5.3.4 Variabilizing C4

The constraint C4 appears in the solution-test as

```
(= (#OCCURRENCES (CLIMAX PROJECT1) PROJECT1))
```

In this form, C4 cannot be moved to the path-test (by RULE323) or path-order (by RULE335), since it is neither monotonically necessary nor sufficient. That is, whether or not the highest note of a partial cantus has occurred only once has no bearing on whether the completed cantus will have the same property. Part of the reason for this is that the two may have different climaxes, since in the course of extending the partial cantus a higher note may be added on.

One way to simplify this problem is to choose a climax tone before generating the cantus (§2.12). In fact, Meehan's program asked the user to select a climax tone [Meehan 72]. This idea can be implemented by variabilizing (CLIMAX (CANTUS)), just as (SUIT-LED) and (CARD-OF ME) were variabilized in the Hearts example:

```
        (= (#OCCURRENCES (CLIMAX PROJECT1) PROJECT1) 1)
13:58   --- [by RULE256] --->
        (AND [= (CLIMAX PROJECT1) CLIMAX1]
             [= (#OCCURRENCES CLIMAX1 PROJECT1)  1])
; ASSUMING (SELECT CLIMAX1 (TONES))
```

Here the variable CLIMAX1 denotes a tone to selected before the search begins. This permits constraint C4 to be split into two sub-goals. The first sub-goal (call it C4a) is to ensure that CLIMAX1 is the climax of the generated cantus (*i.e.*, that no other note is higher). The second sub-goal (C4b) is to make sure that CLIMAX1 occurs exactly once in the cantus, *i.e.*, is not repeated. The idea of restricting a problem by determining one of its features *a priori* is expressed by

RULE256: (P ... (f s) ...) -> (and [= (f s) c] [P ... c ...])
where c is to be selected from (range f) before s is constructed

To construct an object s so as to satisfy a constraint P(s, f(s)),
Choose a value c for f(s).
And solve the two subproblems P(s, c) and f(s) = c.

This strategy may or may not succeed, depending on the choice of c. For example, if CLIMAX1 is chosen to be the lowest tone in (TONES), it will be impossible to construct a legal cantus. Backtracking offers one way around this problem: if the search peters out or continues too long without finding a solution, pick another value for c and start over. A more sophisticated approach would be to identify necessary conditions on the satisfiability of P and choose c accordingly. This approach would avoid choosing the lowest tone in (TONES) for CLIMAX1 because its only legal successor would be a higher note. Whether the detailed analysis required by this approach would be worthwhile would depend on empirical characteristics of the search space: it might well be more efficient to choose CLIMAX1 randomly from the entire set of (TONES) and backtrack when necessary. Another approach is to leave the selection of such parameters to the user, who may know which choices are likely to work.

An alternative is to follow a strategy something like this one, derived in DERIV14 (§2.11):

If the highest note so far occurs only once, try not to repeat it.
Otherwise, try to add a higher note.

Here the importance of "trying" to satisfy a goal increases as the deadline imposed by the end of the cantus approaches. However, it is not clear how to represent the importance of a goal, since FOO lacks a way to refer to the set of goals being pursued at a given time (§8.2.3) -- a consequence of my decision to consider only pieces of advice that can be operationalized independently of each other (§1.1.2).

3.5.3.5 Splitting C4a and Moving it to the Step-test

To move constraint C4a earlier in the search, it is first split into a conjunction:

```
              (= (CLIMAX PROJECT1) CLIMAX1)

       13:1-63 --- [by analysis] --->

              (AND [IN CLIMAX1 PROJECT1]
                   [FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1))])
```

The second conjunct is a necessary condition of the kind discussed earlier, and can be moved from solution-test to path-test to step-test:

```
       13:65-71 --- [by RULE323, analysis] --->
       (HSM1 <- PATH-TEST :
             (AND ... [LAMBDA (PROJECT1)
                        (FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1)))]))

       13:72-80 --- [by RULE327, analysis] --->
       (HSM1 <- STEP-TEST :
             (LAMBDA (I1 NOTE3) (NOT (HIGHER NOTE3 CLIMAX1))))
```

This refinement avoids generating paths containing a note higher than the pre-selected CLIMAX1.

3.5.3.6 Splitting C4b and Moving it to the Step-test

Similarly, splitting constraint C4b into a conjunction allows it to be moved earlier in the search:

```
              (= (#OCCURRENCES CLIMAX1 PROJECT1) 1)

       13:83  --- [by antisymmetric property of >=] --->

              (AND [>= (#OCCURRENCES CLIMAX1 PROJECT1) 1]
                   [>= 1 (#OCCURRENCES CLIMAX1 PROJECT1)])

       13:84-87 --- [by analysis] --->

              (AND [IN CLIMAX1 PROJECT1]
                   [=< (#OCCURRENCES CLIMAX1 PROJECT1) 1])
```

The second clause of this conjunction can be moved into the step-test via the path-test:

```
13:92-104 --- [by RULE323, analysis] --->
(HSM1 <- PATH-TEST :
      (AND ... [LAMBDA (PROJECT1)
                  (=< (#OCCURRENCES CLIMAX1 PROJECT1) 1)]))

13:105-116 --- [by RULE389, analysis] --->
(HSM1 <- STEP-TEST :
      (LAMBDA (I1 NOTE3)
         (AND [NOT (HIGHER NOTE3 CLIMAX1)]
              [OR [NOT (= CLIMAX1 NOTE3)]
                  [NOT (IN CLIMAX1 PROJECT1)]])))
```

The effect of this refinement is to avoid generating a path in which CLIMAX1 is repeated. This is the same kind of transfer performed by RULE327, but uses a slightly different rule:

```
RULE389:    (HSM with    (path-test : (lambda (s) Ps))
                         (step-test : (lambda (i c) Ric))
      ->    (HSM with    (step-test : (lambda (i c) (and Ric Qc))))
where Qc is the result of simplifying (P (append s (list c)))
```

If the path-test contains a constraint P on paths $x_1 \ldots x_k$
And $P(x_1 \ldots x_k)$ is equivalent to $Q(x_k)$,
Then add Q to the step-test.

Although RULE389 is not used in the "avoid taking points" example, it closely resembles in form the rule used to move the have-points constraint from path-order to step-order:

```
RULE338:    (HSM with    (path-order : (lambda (s) Ps)))
      ->    (HSM with    (step-order : (lambda (i c) Qc))),
where Qc is the result of simplifying (P (append s (list c)))
```

If the path-order contains a constraint P on paths $x_1 \ldots x_k$
And $P(x_1 \ldots x_k)$ is equivalent to $Q(x_k)$,
Then add Q to the step-order.

The analysis used to simplify the step-test version of constraint C4b includes the interesting step

```
         (=< (#OCCURRENCES CLIMAX1 PROJECT1) 1)
13:108 --- [EVAL by RULE390] --->
         T
```

This step is based on the inductive argument that the step-test will only be used to extend paths that already satisfy the path-test, so any property of PROJECT1 required by the path-test can be assumed true in the step-test. This idea is expressed in general as

```
RULE390:    (HSM with    (path-test : (lambda (s) (and ... Ps ...)))
                         (step-test : (... Ps ...)))
      ->    (HSM with    (step-test : (... T ...)))
```

Remove a constraint Ps from the step-test if it occurs in the path-test.

3.5.4. Refined HSM Formulation for Cantus Firmus Example

The net result of the refinements for the cantus example is to instantiate the HSM schema as follows:

```
(HSM1 WITH
      (INITIAL-PATH : NIL)
      (CHOICE-SETS :
         (LAMBDA (I1)
            (IF (= I1 1)
                (TONES)
                (USABLE-SUCCESSORS-OF (PROJECT1 (- I1 1))))))
; USABLE-SUCCESSORS-OF =
    (LAMBDA (NOTE1)
       (SET-OF NOTE2 (TONES)
          (USABLE-INTERVAL (INTERVAL NOTE1 NOTE2))))

      (STEP-ORDER : NIL)
      (STEP-TEST :
         (LAMBDA (I1 NOTE3)
            (AND [NOT (HIGHER NOTE3 CLIMAX1)]
                 [OR [NOT (= CLIMAX1 NOTE3)]
                     [NOT (IN CLIMAX1 PROJECT1)]]])))

      (PATH-ORDER : NIL)
      (PATH-TEST :
         (AND [LAMBDA (PROJECT1)
                 (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                    (USABLE-INTERVAL INTERVAL1))]
              [LAMBDA (PROJECT1)
                 (=< (MELODIC-RANGE PROJECT1) (MAJOR 10))]
              [LAMBDA (PROJECT1)
                 (FORALL X1 PROJECT1
                    (NOT (HIGHER X1 CLIMAX1)))]
              [LAMBDA (PROJECT1)
                 (=< (#OCCURRENCES CLIMAX1 PROJECT1) 1)]))

      (COMPLETION-TEST :
         (LAMBDA (PROJECT1) (IN (LENGTH PROJECT1) (LB:UB 8 16))))
      (SOLUTION-TEST :
         (LAMBDA (PROJECT1)
            (AND [IN CLIMAX1 PROJECT1]
                 [=< (#OCCURRENCES CLIMAX1 PROJECT1) 1]
                 [FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1))]
                 [FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                    (USABLE-INTERVAL INTERVAL1)]
                 [=< (MELODIC-RANGE PROJECT1) (MAJOR 10)]))))

; ASSUMING (SELECT CLIMAX1 (RANGE (CLIMAX PROJECT1)))
```

The search thereby specified corresponds to the refined procedure shown in Figure 3-8, and has the following features.

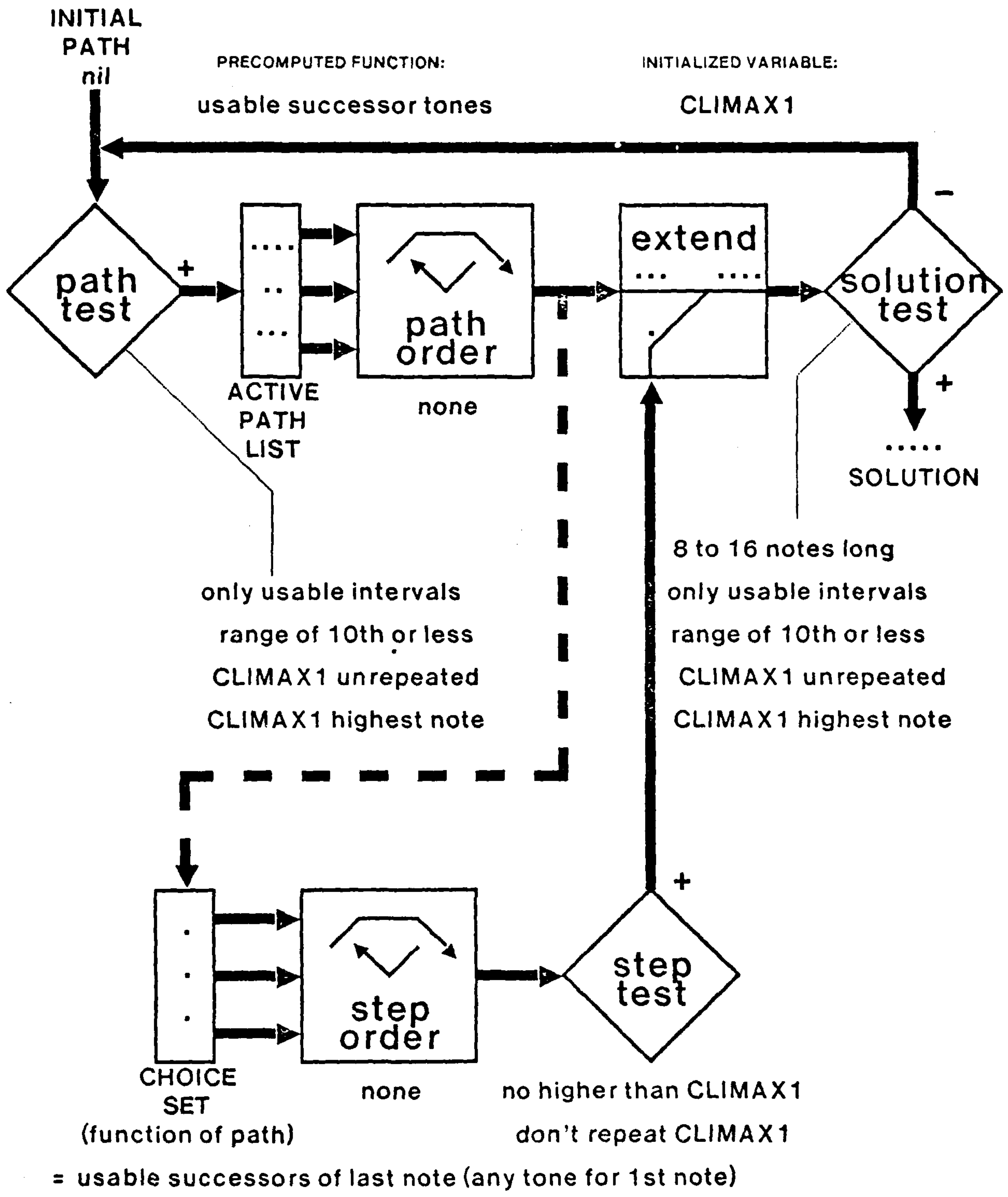

Figure 3-8: Refined Search for Compose Cantus

The object of the search is to find a tone sequence, indexed by integer, satisfying the solution-test and completion-test.

The completion-test specifies that

C1. The cantus be from 8 to 16 notes in length.

The solution-test specifies that

C2. Every interval of the cantus satisfy the `USABLE-INTERVAL` predicate.

C3. The melodic range of the cantus not exceed a major tenth.

C4. The preselected climax `CLIMAX1` occur exactly once in the cantus.

The choice set at each point in the search, indexed by integer variable `I1`, consists of the `USABLE-SUCCESSORS-OF` the previous note, except that the initial choice set includes the whole set of tones. The usable successors of each tone can be precomputed.

The search proceeds by choosing a path from the active path list, extending it by one element, testing the result to see if it's a solution, and deciding whether to add it to the active list. The first path in the active list is the empty sequence.

The step-test is used to test possible extensions, so that sequences containing notes higher than `CLIMAX1`, or more than one occurrence of `CLIMAX1`, are not even considered.

The path-test is used to test generated paths, so that sequences whose melodic range exceeds a major tenth are rejected.

The solution-test is used to test possible cantuses; sequences not containing `CLIMAX1` are rejected.

The search terminates successfully when it finds a sequence satisfying both the solution-test and the completion-test. By the construction of the search, any such sequence will satisfy C1-C4.

3.5.5. Other Possible Refinements

The cantus firmus example was undertaken to test the generality of the HSM rules used in the "*avoid taking points*" example. It used the same rules for initial instantiation of the HSM schema, and some of the same rules for refining the search. It is instructive to look at the other HSM refinement rules used in the "avoid taking points" example and consider whether they too could have been applied to the cantus example. Similarly, one may ask whether the rules added to account for particular refinements could be used to make other refinements as well, such as those that appear in Meehan's solution. This section discusses refinements that were not included in the cantus example but could have been.

3.5.5.1 Changing the Initial-path

The music textbook from which constraints C1-C4 were drawn presents several other constraints on cantus firmus. One of these is:

"The cantus firmus ... must begin and end on the tonic."

This constraint could be added to the definition of LEGAL-CANTUS:

```
(LAMBDA (S)
   (AND <previous definition of LEGAL-CANTUS>
        [= (FIRST S) (TONIC)]
        [= (LAST S) (TONIC)]))
```

The constraint could be moved from solution-test to step-test by RULE323 and RULE327. The first-note-is-tonic constraint could then be compiled out of the search by

RULE332: *Start the search at the point following any choices determined before search time.*

This would change the initial path to consist of the tonic note:

```
(HSM1 WITH
      (STEP-TEST :
         (LAMBDA (I1 NOTE1)
            (AND ... [=> [= I1 1] [= NOTE1 (TONIC)]] ...))))

--- [by RULE332, analysis] --->
(HSM1 <- INITIAL-PATH : (LIST NOTE2))
(HSM1 <- BINDINGS : ((TONIC)))
(HSM1 <- VARIABLES : (NOTE2))
```

The same rule was used in the "avoid taking points" example to change the initial path to the [illegible] of cards played by the players up to and including player ME (§3.4.2).

3.5.5.2 Variabilizing C1

Meehan's program simplified constraint C1 on the length of the cantus by asking the user to specify the length of the cantus *a priori*. A similar effect could be achieved by applying a rule for preselecting a problem feature and a rule for constraining choices:

RULE256: *To construct an object s so as to satisfy a constraint P(s, f(s)), Choose a value c for f(s), And solve the two subproblems P(s, c) and f(s) = c.*

RULE156: *To achieve P(... c ...) where c is chosen from S, choose c from {x in S / P(... x ...)}.*

These rules would be applied to the cantus-length constraint in the solution-test:

```
(IN (LENGTH PROJECT1) (LB:UB 8 16))

--- [by RULE256, analysis] --->
(AND [IN LENGTH1 (LB:UB 8 16)]
     [= (LENGTH PROJECT1) LENGTH1])
; ASSUMING (SELECT LENGTH1 (INTEGERS))

--- [by choice constraint, analysis] --->
(HSM1 <- COMPLETION-TEST :
            (LAMBDA (PROJECT1)
               (= (LENGTH PROJECT1) LENGTH1)))
; ASSUMING (SELECT LENGTH1 (LB:UB 8 16))
```

This refinement specifies that LENGTH1 is chosen before the search. The choice could be made randomly or by asking the user for a number between 8 and 16.

3.5.5.3 Using C4 to Order the Search

The cantus example does not include step-order or path-order criteria for directing the search as in the Hearts example, but such criteria could be derived using the same rules:

RULE335: *If the solution-test includes a monotonically sufficient constraint P Then add P to the path-order.*

RULE338: *If the path-order contains a constraint P on paths $x_l \ldots x_k$ And $P(x_l \ldots x_k)$ is equivalent to $Q(x_k)$, Then add Q to the step-order.*

These rules were used in the "avoid taking points" example to re-order the search so as to consider card sequences with points first. In the cantus example, they could be applied to a clause of the unique-climax constraint C4 in the solution-test:

```
    (IN CLIMAX1 PROJECT1)
```

This clause would be moved into the step-order in the form

```
    (LAMBDA (I1 NOTE1)
        (OR [IN CLIMAX1 PROJECT1]
            [= NOTE1 CLIMAX1]))
```

This would reorder the search to use CLIMAX1 first when extending a note sequence in which it does not occur. However, this would produce the musically undesirable effect of always selecting CLIMAX1 as the first tone of the cantus. Actually, this effect would not be produced by a faithful rendering of the original textbook version of C4:

"Each cantus firmus must contain a climax or high point. *This tone will serve as the goal of a motion from the first tone of the cantus; it will simultaneously function as the beginning of a second melodic curve down to the final tone.* The climax tone should not be repeated."

In encoding C4 I omitted the italicized sentence because it wasn't obvious how to formalize such vague concepts as "the goal of a motion" and "melodic curve" without reading extra meaning into the text and thereby doing a lot of operationalization myself. Incorporating additional textbook constraints in the definition of LEGAL-CANTUS would moderate the effect of using C4 to order the search and prevent the unmusical selection of CLIMAX1 as the first note. Even then, using C4 to order the search might be counter-productive, since it would tend to include CLIMAX1 as early as possible in the cantus, contrary to aesthetic considerations. This refinement is heuristic in nature: its utility would depend strongly on the other constraints in the search criterion and on the other refinements made. Given the definition used in the example, such a re-ordering is a legitimate refinement since it would avoid considering complete paths in which CLIMAX1 did not occur.

3.5.5.4 Converting C3 to a Step-constraint

One could use C3 to order the search by preferring extensions that do not increase the melodic range. This preference might be derived mechanically in the form of a step-order as follows:

```
    (HSM1 WITH
          (PATH-TEST :
             (LAMBDA (PROJECT1)
                (AND [=< (MELODIC-RANGE PROJECT1) (MAJOR 10)] ...))))

    --- [by RULE338a] --->
    (HSM1 <- STEP-ORDER :
          (LAMBDA (I1 NOTE1)
             (=> [=< (MELODIC-RANGE PROJECT1) (MAJOR 10)]
                 [=< (MELODIC-RANGE
                        (APPEND PROJECT1 (LIST NOTE1)))
                     (MAJOR 10)])))
          --- [SUFFICE by RULE383] --->
          (LAMBDA (I1 NOTE1)
             (=< (MELODIC-RANGE (APPEND PROJECT1 (LIST NOTE1)))
                 (MELODIC-RANGE PROJECT1)))

    --- [by analysis] --->
    (HSM1 <- STEP-ORDER :
          (LAMBDA (I1 NOTE1)
             (AND [NOT (LOWER NOTE1 (LOWEST PROJECT1))]
                  [NOT (HIGHER NOTE1 (HIGHEST PROJECT1))])))
```

This refinement could be accounted for by a rule like

RULE338a: *If the path-test contains a constraint P on paths $x_1 \ldots x_k$,*
And the inductive condition $P(x_1 \ldots x_{k-1}) => P(x_1 \ldots x_k)$ reduces to $Q(x_k)$,
Then add Q to the step-order.

This hypothetical rule strongly resembles a rule used in the Hearts example:

RULE338: *If the path-order contains a constraint P on paths $x_1 \ldots x_k$,*
And $P(x_1 \ldots x_k)$ is equivalent to $Q(x_k)$,
Then add Q to the step-order.

The effect of this refinement would be to prefer to extend a sequence without extending its melodic range, which is not strictly necessary to stay within the bounds specified by C3.

The exact condition for C3, formulated as a step-test, would be

```
    (LAMBDA (I1 NOTE1)
       (AND [=< (SIZE (INTERVAL (LOWEST PROJECT1) NOTE1)) (MAJOR 10)]
            [=< (SIZE (INTERVAL NOTE1 (HIGHEST PROJECT1)) (MAJOR 10)]))
```

The effect of this step-test would be to avoid choosing a note more than a major tenth above the lowest note so far or more than a major tenth below the highest note so far. Deriving this condition mechanically appears to require expanding the definition of MELODIC-RANGE down to a low level of detail, or else using a fair amount of specific knowledge about interval arithmetic.

3.5.5.5 Variabilizing and Precomputing C3

It is interesting to note that Meehan simplified constraint C3 in his program by asking the user to specify *a priori* the lowest and highest permissible tones, and then restricted his generator to the interval spanned by them. FOO's refinement rules are nearly adequate to derive this solution:

```
    (=< (MELODIC-RANGE PROJECT1) (MAJOR 10))
    --- [by RULE256', antisymmetry of =<] --->
```

```
(AND [=< MELODIC-RANGE1 (MAJOR 10)]
     [=< (MELODIC-RANGE PROJECT1) MELODIC-RANGE1])
; ASSUMING (SELECT MELODIC-RANGE1 (INTEGERS))
```

This transformation uses a variant of the rule that suggested pre-selecting the climax tone:

RULE256a: *To construct an object s so as to satisfy a constraint P(s,f(s)),*
Choose a value c for f(s),
And solve the two problems P(s, c) => P(s, f(s)) and f(s) = c.

The idea here is to choose MELODIC-RANGE1 *a priori* and then use it to limit the actual range of the generated cantus. This goal can be achieved as follows:

```
(=< (MELODIC-RANGE PROJECT1) MELODIC-RANGE1)

--- [by RULE124, RULE271a] --->
(=< (SIZE (INTERVAL (LOWEST PROJECT1) (HIGHEST PROJECT1)))
    (SIZE INTERVAL1))
; MELODIC-RANGE1 <- (SIZE INTERVAL1)

--- [by analysis, RULE271a] --->
(SUBSET (INTERVAL (LOWEST PROJECT1) (HIGHEST PROJECT1))
        (INTERVAL LOWEST1 HIGHEST1))
; INTERVAL1 <- (INTERVAL LOWEST1 HIGHEST1)

--- [by analysis] --->
(AND [NOT (LOWER (LOWEST PROJECT1) LOWEST1)]
     [NOT (HIGHER (HIGHEST PROJECT1) HIGHEST1)])
```

Here RULE271a can be paraphrased as:

Assume an unbound variable compared to an expression can be expressed in a similar form.

This idea is expressed more precisely (§5.4.2) as

RULE271a: (R [f e_1 ... e_n] e) -> (R [f e_1 ... e_n] [f v_1 ... v_n]), assuming e = (f v_1 ... v_n)

Thus simplified, C3 can be moved from the solution-test to the path-test by RULE323, then to the step-test by RULE327 and some analysis based on the transitivity of LOWER and HIGHER:

```
(HSM1 <- STEP-TEST :
    (LAMBDA (I1 NOTE1)
       (AND [NOT (LOWER NOTE1 LOWEST1)]
            [NOT (HIGHER NOTE1 HIGHEST1)])))
```

C3 can then be moved from the step-test into the choice-set by RULE384, and replaced with a precomputable function by RULE386 (which suggested precomputing USABLE-SUCCESSORS-OF):

```
(HSM1 <- CHOICE-SETS :
    (LAMBDA (I1)
       (TONES-BETWEEN LOWEST1 HIGHEST1)))
```

The precomputable function synthesized in this case would be

```
TONES-BETWEEN =
  (LAMBDA (NOTE1 NOTE2)
     (SET-OF NOTE3 (TONES)
        (AND [NOT (LOWER NOTE3 NOTE1)]
             [NOT (HIGHER NOTE3 NOTE2)])))
```

This function could be precomputed and stored in a two-dimensional table, or computed quickly at search time by a loop of the form

```
For NOTE3 from NOTE1 to NOTE2
Do <try using NOTE3 to extend the current path>
```

Meehan's program uses this kind of loop to generate candidates for the next note. Adding rules to FOO on using iteration to generate choice sets would probably involve defining an explicit iteration construct and representing the concept of successor function.

3.5.5.6 Satisfying C4 without Predetermining the Climax

One limitation of the solution in the cantus example (and of Meehan's program) is the requirement that the climax tone be pre-selected. The mechanical derivation of a strategy for satisfying C4 without choosing the climax *a priori* is demonstrated in DERIV14 (§2.11). DERIV14 treats C4 in terms of temporal goals, where the sole action consists of extending the cantus by another note. A general rule for achieving a goal over time is

RULE258: (achieve P) -> (and [=> [not P] [cause P]] [=> P [not (undo P)]])

Achieve a goal by making it true if it isn't, and not undoing it if it is.

Translating this goal-oriented rule into the HSM paradigm yields a refinement rule:

```
RULE258a:    (HSM with     (solution-test : (lambda (s) (and ... [P s] ...))))
       ->    (HSM with     (step-order : (lambda (i c) (and Qc Rc))))
where Qc is the result of simplifying (=> [P s] [P (append s (list c))])
and Rc is the result of simplifying (=> [not (P s)] [P (append s (list c))])
```

If the solution-test contains a constraint P on a path $x_1 \ldots x_n$
And $P(x_1 \ldots x_{k-1}) \Rightarrow P(x_1 \ldots x_k)$ *reduces to* $Q(x_k)$
And $\sim P(x_1 \ldots x_{k-1}) \Rightarrow P(x_1 \ldots x_k)$ *reduces to* $R(x_k)$.
Then put Q and R into the step-order.

This rule is not as logically rigorous as the rules *about montonically* necessary and sufficient conditions; it assumes that extensions of a path satisfying some constraint tend to satisfy it also. This assumption of "inertia" or "monotonicity" has obvious counter-examples. For instance, if you want to give a dinner party next week and need to defrost your roast first, taking it out of the freezer today is unlikely to help you achieve your goal of having a good party. Or if you want to leave on a bicycle trip tomorrow, you will need to unlock your bicycle and move it in front of your house, but it may not stay there overnight if you do it now. Thus the monotonicity assumption can fail in domains where there are other agents, *e.g.*, bacteria and bicycle thieves.

Even if the constraint in question continues to hold when the path is extended, a refinement may be bad because it interferes with other constraints. As mentioned earlier, re-ordering the search to use the climax tone as early as possible violates aesthetic criteria on the melodic contour of the cantus. Similarly, playing the Queen of Spades at the first opportunity when trying to shoot the moon in Hearts violates the goal of concealing one's intentions from opponents, even though the subgoal of taking the Queen will remain fulfilled.

3.5.5.7 Depth-first versus Breadth-first Search

A *depth-first* search order could be achieved by changing the initial path-order to `(LAMBDA (S) (LONG S))`. Here the fuzzy predicate `LONG` returns a value proportional to the length of the path. This assumes that the mechanism that evaluates operational expressions at runtime can handle fuzzy logic. For example, fuzzy truth values could be represented as real numbers between 0 and 1, and combined arithmetically to compute boolean combinations. A disjunction of fuzzy truth values might be evaluated by taking their maximum. Other boolean operators might correspond to minimum, complement, sum, product, average, etc.

Best-first behavior might be achieved by combining the predicate `LONG` with domain-specific criteria based on monontonically sufficient conditions derived from the solution-test in the manner illustrated in the examples. This would focus the search on well-developed partial paths satisfying several constraints.

Conversely, *breadth-first* search could be implemented giving equal priority to all branches of the search tree. This could be achieved by incorporating in the path-order the fuzzy predicate `(LAMBDA (S) (SHORT S))`. This option appears inappropriate for either the Hearts or the music example. Breadth-first behavior decreases the risk of squandering limited resources in exploring the extensions of a poor choice made early in the choice sequence, at the cost of slowing down progress toward a solution by cautiously examining inferior alternatives along the way.

An interesting direction for future research is to characterize the situations in which each of these alternatives is most effective. For example, breadth-first search may be appropriate when solutions are sparsely distributed in the search space and there are no good heuristics for evaluating the relative promise of a given path, *i.e.*, predicting whether it can be extended into a solution.

3.6. Evaluation of the Experiment

Attacking a problem in a second task domain provided a way to test the generality of FOO's knowledge about operationalizing advice in terms of HSM, and the overall viability of the approach. The generality issue divides into three parts corresponding to different parts of this knowledge:

1. The HSM schema itself.
2. The rules for instantiating the HSM schema initially.
3. The rules for refining the search.

3.6.1. Generality of the HSM Schema

The same schema was used for both the Hearts and the music examples. Some components were added in the course of developing the first example, like the initial-path and variables, but they proved useful in the second problem as well. It would be necessary to add or generalize components to handle certain other problems -- for instance problems in which the initial state is a set of paths rather than a single path. For some problems it would be useful to have a test applied only to the initial-path, or to specialize the step-constraints into different predicates based on the arguments they require. For instance, some of the tests depend on the path variable, and some only on the element to be used for extending the path. Presumably the latter are cheaper to evaluate than the former.

Endless variations on the generic search procedure are possible, and each one has cases where it would be useful. It would be interesting (and no doubt quite challenging!) to mechanize the analytic process involved in inventing such modifications. A prerequisite of such an effort is a way to represent modifications to HSM. MERLIN was able to represent some of these as "further specifications" of an abstract description [Moore 74]. FOO's schema has the advantage of offering a fixed target for a set of instantiation and refinement rules, and appears quite adequate to handle both the examples presented in this chapter. However, using a fixed collection of control components precludes the use of graph transformations, such as splitting the search procedure into several copies and specializing each one to handle a separate case. A more powerful approach would represent refinements as transformations on flow graphs. This approach has been used to design efficient algorithms for generating prime numbers and finding shortest paths through a graph [Tappel 80], and appears to be a promising avenue for future research.

3.6.2. Generality of the Schema Instantiation Process

The process of initially instantiating the HSM schema is almost the same in the two examples, step for step: the same HSM rules are applied in the same order, although the analysis required in between often differs substantially. The only deviation from the common path occurs in the process of fixing up the initial instantiation to make it executable. In the Hearts example, this consists of broadening the choice set to make it evaluable, and then finding evaluable predicates with which to filter it back down (§3.3.4). The rules used to fix up the Hearts example do not apply in an obvious way to the music example, but appear related more to the problem of reformulation-for-evaluability than to HSM *per se*. In the music example, the only fix is to replace an undefined completion-test with a problem constraint (§3.5.1) in order to compensate for a limitation of FOO's representation scheme.

The striking parallelism between the instantiation process in the two examples provides supporting evidence for the generality of that process.

3.6.3. Generality of the Refinement Process

The generality of the refinement rules developed for the Hearts example was tested by checking to see which of them could be applied to the music example. Some of them, like RULE323 and RULE327, were directly and repeatedly applied in the music example. Others are very similar to rules used in the music example: RULE389 resembles RULE338 in form, and RULE333 resembles RULE256 in spirit. Finally, some rules, like RULE332 and RULE335, were not actually used in the new example but could have been. By these criteria, *every refinement rule used in the Hearts example was (or resembled a rule) applicable to the music example.* The cases of close resemblance between rules suggest that it may be worthwhile to look either for common generalizations or for a more fundamental rule-generating process.

Conversely, one can ask how many refinement rules were added to cover the music example, not counting rules that were almost the same as old ones. The new rules were:

RULE256 for splitting a constraint into choose now--solve later form, which resembles in spirit RULE333 for variabilization;

RULE384 for moving a step-test into a generator;

RULE386 for recognizing precomputable functions, an idea not specific to refining HSM;

RULE389 for moving a path-test to a step-test, which resembles in form the old RULE338 for moving a path-order to a step-order; and

RULE390, which uses an inductive property of the search to simplify step-constraints, and is really an analysis rule that expresses a fact about HSM.

However, none of these rules appears specific to music. I did not explore the possibility of applying them to the Hearts example.

According to the criteria above -- applicability or similarity -- *all* the refinement rules used in the Hearts example were general within the scope of the two examples, and the ones added to handle the music example were similar to old rules or of general interest in their own right. In contrast, many additional analysis rules were added for the music example to solve the subproblems generated by the HSM rules.

3.6.4. Conclusions on Generality

The experiment provides about as much evidence as one could hope to get from just two examples for the generality of FOO's knowledge about HSM and the process of applying it to problems in a given task domain. The rules that applied to one example but not the other appear to fall in three categories:

1. Rules that deal with problem-specific artifacts unrelated to HSM *per se*, such as changing the choice set to make it evaluable.
2. Rules whose purpose is not specifically "operationalization in terms of HSM," *e.g.*, rules to perform analysis or identify precomputable functions.
3. Rules similar in form or purpose to rules used in the other example, but syntactically or functionally not general enough to be used directly in the other example, *e.g.*, a rule that moves constraints from path-test to step-test but not from path-order to step-order.

The fact that the HSM rules used in the two examples overlapped to such a great extent and were used more than once within the same example, while the analysis problems they generated required dissimilar rules, suggests that FOO does indeed have some general knowledge about the instantiation and refinement processes involved in operationalizing a problem in terms of HSM.

Chapter 4
Reformulation

The most common activity exhibited in the example derivations is the reformulation of expressions to fit the run-time capabilities of the task agent, or the requirements of a particular method. Often this is only implicit in the directionality of the derivation: a sequence of reformulation steps precedes the application of a significant operationalization or problem reduction method, but the goal "reformulate the expression so this method can be applied" is not *explicitly* represented in the derivation.

The rest of this chapter is organized according to different kinds of reformulation *problems*, rather than the *methods* used to solve them. (§4.1) explains the difficulty of representing such problems explicitly. (§4.2) describes two important classes of explicit reformulation problems. (§4.3) discusses several kinds of reformulation problems implicit in the derivations. Finally, (§4.4) re-examines the explicit/implicit distinction and presents a taxonomy of reformulation problems.

4.1. Representation of Literal Expressions

Consider the problem of *explicitly* representing a reformulation goal like

"Transform (GET-LEAD ME) into an equality."

This goal refers to the *literal expression* (GET-LEAD ME) as opposed to the action it denotes. However, the knowledge representation used in FOO provides no way to refer to the *form* of literal expressions; expressions are considered semantically equivalent if they denote the same entity. This limitation applies only to the expression language, not to the rules. A rule condition can test whether an expression has a particular form, either by matching it against a variabilized pattern or by using a procedure to examine its internal representation. However, a *predicate* on the form of an expression would require as its argument some representation of the literal expression. In LISP, the literal object *e* is denoted '*e*. This construct might be used to represent the goal of transforming (GET-LEAD ME) into an equality:

"Find P s.t. (Interpretation P) = (Interpretation '`(GET-LEAD ME)`) and (Car P) = '=."

Here the variable P takes as its value a literal expression, whose meaning, denoted by (Interpretation P), is required to be semantically equivalent to `(GET-LEAD ME)`. Representing a constraint on the *form* of an expression may require knowledge about how expressions are encoded in the machine. Thus the requirement that P be an equality is expressed as (Car P) = '=, where '= denotes the literal symbol for equality. This assumes that expressions are encoded in LISP prefix notation. However, representing literal expressions has the disadvantage of violating the referential transparency of the representation. Moreover, the 'e construct does not handle the case where e has variables, *e.g.*, `(GET-LEAD P1)`. Similar difficulties arise when one tries to represent such notions as "knowing that X is true" or "knowing the value of X." Modal logic has been used to represent the "knowing" relation [McCarthy 77].

4.2. Explicit Reformulation Problems

It is possible to represent explicitly certain important classes of reformulation problems without having to represent literal expressions and operations on their structure. One such class contains problems of the form

Reformulate expression e in terms of concept f.

These problems can be represented as equations of the form

$$e = h(f(x_1, \ldots, x_k))$$

In these equations, the function h and arguments $x_1, \ldots, x_k$ are the unknowns. Another class contains problems of the form

Reformulate expression e as a function of quantity e'.

These can be represented as equations of the form

$$e = h(e')$$

Here the function h is the unknown. For both classes of reformulation problems, the equation is solved and the unknowns are replaced by their values to yield an expression of the desired form.

4.2.1. Reformulating an Expression in terms of a Given Concept

The first kind of reformulation problem has the form

Reformulate expression e in terms of concept f.

Such a problem is illustrated in DERIV9. Here, f is the predicate DISJOINT and e is

```
    (EXISTS C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0))
```

Since this expression is a predicate on the unknown set (CARDS-IN-HAND P0), it makes sense to reformulate it in terms of the predicate DISJOINT to match the problem statement for the disjoint subsets method (§2.3). The reformulation problem is set up as an explicit equation:

```
    9:2 --- [by RULE189] --->
    (= (EXISTS C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0))
       (H1 (DISJOINT (CARDS-IN-HAND P0) S1)))
```

The first step toward solving for H1 is to plug in the definition of DISJOINT:

```
    9:3-4 --- [ELABORATE by RULE124] --->
    (= (EXISTS C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0))
       (H1 (NOT (EXISTS X1 (CARDS-IN-HAND P0) (IN X1 S1)))))
```

If the two terms of the form (EXISTS ...) can be proved equal, H1 can be solved for by

RULE212: (= e (h (... e' ...))) -> (= h (inverse (lambda (x) (... x ...)))) if e = e'

A value of S1 is found that makes the two terms equal:

```
    (= (EXISTS C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0))
       (EXISTS X1 (CARDS-IN-HAND P0) (IN X1 S1)))
    9:6-13 --- [by analysis] --->
    T
    (S1 <- BINDING : (CARDS-IN-SUIT S0))
```

The recurrence rule RULE212 (§5.3.5) gives a solution (there may be others) for the function H1:

```
    (= H1 (INVERSE (LAMBDA (EXISTS1) (NOT EXISTS1))))
    9:15 --- [simplify] --->
    (= H1 (INVERSE NOT))
```

This yields the function NOT (which happens to be its own inverse) as the value of H1.

4.2.2. Reformulating an Expression as a Function of a Given Quantity

A second class of reformulation problems have the form

Reformulate expression e as a function of quantity e'.

Such problems are illustrated in DERIV2 and DERIV13 (§3). Once the CHOICES slot of the instantiated HSM has been filled in with the sequence of objects to be chosen, RULE314 proposes to reformulate the initial problem as a function of this sequence. In DERIV2, this is the sequence (CARDS-PLAYED) of cards played in the trick; in DERIV13, the sequence consists of the notes in the cantus. For convenience, the derivations use the notation (reformulate e e') rather than the equation e = *h*(e'). Once the problem is massaged into the form (reformulate (f ... e' ...) e'), RULE315 notices the recurrence on e' and reformulates e as ([lambda (x) (f ... x ...)] e').

Problems of the form "reformulate P as a predicate Q universally quantified over the set of path indices" are generated by RULE327 at steps 2:74, 13:28, and 13:72 in the course of moving constraints from the path-test to the step-test (§3.4.1). Each such problem is represented as an equation of the form P = (forall i (indices-of path) Q).

A somewhat similar problem is generated in DERIV9 by the decision to approximate the probability that player P0 is void in suit S0 as a function of S0 (§2.8). This problem has the form

Reformulate expression e as a 0th-order approximate function of e'

The problem (generated by RULE202 at step 9:44) is not represented as an equation of the form e ~ *h*(e'). Instead, RULE203-208 are used to compute (dependence e e') -- essentially the sign *d* of the partial derivative of e with respect to e' in the 4-element algebra {↑, ↓, 0, ?} -- and e is rewritten as (FUNCTION-OF e' *d*). A slight complication arises because it is impossible to partial-differentiate with respect to a non-continuous variable, in this case the suit S0. This is fixed up by RULE210, which finds a continuous function e" of e', and changes the problem to computing *d'* = (DEPENDENCE e e") and approximating e as (FUNCTION-OF e" *d'*). The final expression is

```
(FUNCTION-OF (#CARDS-OUT-IN-SUIT S0) DECREASING)
```

It can be interpreted as "a decreasing function of the number of cards out in suit S0." This gives enough information to order the suits according to the relative probabilities of a void in each one. The ordering could be computed by the task agent's run-time evaluation mechanism (given a way of computing (#CARDS-OUT-IN-SUIT S0) as derived in DERIV10 (§2.4.3)) and used in applying advice like "Avoid leading a high card in a suit where an opponent is void."

4.2.2.1 Identifying Potentially Applicable Methods

A problem-solver in the operationalization problem space would need to know when reformulation was appropriate. In some cases, reformulation is explicitly called for by a specific strategy based on a clue in the problem, *e.g.*, "use the disjoint subsets method to estimate a predicate on an unknown set." In other cases, reformulating a problem makes it possible to apply some method to it. It would be desirable to have an efficient way to identify problems potentially amenable to a known useful method. One approach would partial-match an expression (f ...) against the left-hand side of a method, and use a near-match as a clue to the potential applicability of the method. Residual mismatches would then be resolved by reformulating (f ...) to fit the method. This approach could be expected to fail for top-level mismatches, *e.g.*, where matching the method requires elaborating f. Intersection search through the knowledge base might help identify cases in which such top-level reformulation enables application of a method. A similar approach is described in (§7); the general issue of problem-solving in operationalization is discussed in (§8.1.1).

4.3. Implicit Reformulation Problems

Implicit reformulation problems occur throughout the derivations, but seem to fall into a small number of categories. One type of problem involves *reformulating an expression to fit* the task agent's run-time capabilities for making observations, performing actions, and making choices. Another involves mapping an expression into the form required by a method.

4.3.1. Reformulating an Expression to Fit the Run-time Capabilities of the Task Agent

The task agent is assumed to have run-time capabilities for *observing* certain quantities in the task environment, *computing* certain functions, and *executing* certain actions in the environment (§1.1.1).[8] Accordingly, operationalizing an expression may involve reformulating it as an *observable quantity*, a *computable function*, or an *executable action*.

[8]The interactive mode of operation made it unnecessary to give FOO an explicit enumeration of run-time capabilities; the derivations implicitly reflect the knowledge of these capabilities that I used in choosing which rules to apply. This is only one example of the savings in work afforded by factoring problem-solving issues out of the investigation.

4.3.1.1 Reformulating an Expression in terms of Observables

In DERIV4, the pigeon-hole principle is used to reformulate (QS-OUT) in terms of evaluable quantities. Part of the derivation consists of mapping the result of the method to the observable predicates IN-POT and HAS-ME:

```
                    (= (LOC QS) POT)
4:35                --- [RECOGNIZE by RULE123] --->
                    (AT QS POT)
4:40                --- [RECOGNIZE by RULE123] --->
                    (IN-POT QS)

                    (= (LOC QS) (HAND ME))
4:37                --- [RECOGNIZE by RULE123] --->
                    (AT QS (HAND ME))
4:41                --- [RECOGNIZE by RULE123] --->
                    (HAS ME QS)
4:42                --- [RECOGNIZE by RULE123] --->
                    (HAS-ME QS)
```

Later, the abstract action "cause to be at" is reformulated as the observable action TAKE:

```
                    (CAUSE (AT QS (PILE P3)))
4:51                --- [RECOGNIZE by RULE358] --->
                    (MOVE QS LOC1 (PILE P3))
4:52                --- [RECOGNIZE by RULE123] --->
                    (TAKE P3 QS)
```

Reformulation in terms of observable events is described in (§2.4). This kind of transformation is also used in DERIV10 to reformulate "undo out" as the observable action PLAY:

```
                         (UNDO (OUT X1))

                               (OUT X1)
4:26                           --- [ELABORATE by RULE124] --->
                               (EXISTS P3 (OPPONENTS ME) (HAS P3 X1))

                         (UNDO (EXISTS P3 (OPPONENTS ME) (HAS P3 X1)))
4:27                     --- [UNCERTAIN by RULE329] --->
                         (EXISTS P3 (OPPONENTS ME) (UNDO (HAS P3 X1)))

                                       (HAS P3 X1)
4:28                                   --- [ELABORATE by RULE124] --->
                                       (AT X1 (HAND P3))

                                (UNDO (AT X1 (HAND P3)))
4:29                            --- [RECOGNIZE by RULE368] --->
                                (MOVE X1 (HAND P3) LOC2)

4:30                            --- [RECOGNIZE by RULE123] --->
                                (PLAY P3 X1)

                         (EXISTS P3 (OPPONENTS ME) (PLAY P3 X1)))
```

In the examples above, expressions were reformulated in terms of observable conditions and actions. An expression might also be reformulated in terms of an observable state function other than a boolean condition, for example, (CARDS-IN-HAND ME), whose value is a set of cards.

4.3.1.2 Reformulating a Goal as a Function of a Choice Variable

One way to operationalize a goal is to reformulate it as a computable predicate on a choice variable. This predicate can then be incorporated in an evaluation function used for choosing an optimal value for the variable.

For instance, most of the work in DERIV6 consists of reformulating "avoid taking points" as a constraint on the card player ME chooses to play:

```
(AVOID-TAKING-POINTS (TRICK))

6:1 --- [ELABORATE by RULE124] --->
(AVOID (TAKE-POINTS ME) (TRICK))

6:2 --- [ELABORATE by RULE124] --->
(ACHIEVE (NOT (DURING (TRICK) (TAKE-POINTS ME))))

                       (TRICK)
6:3                    --- [ELABORATE by RULE124] --->
                       (SCENARIO (EACH P1 (PLAYERS) (PLAY-CARD P1))
                                 (TAKE-TRICK (TRICK-WINNER)))

               (DURING (SCENARIO (EACH P1 (PLAYERS) (PLAY-CARD P1))
                                 (TAKE-TRICK (TRICK-WINNER)))
                       (TAKE-POINTS ME))
6:4-6          --- [by case analysis] --->
               (DURING (TAKE-TRICK (TRICK-WINNER))
                       (TAKE-POINTS ME))

                       (TAKE-POINTS ME)
6:7                    --- [ELABORATE by RULE124] --->
                       (FOR-SOME C1 (POINT-CARDS) (TAKE ME C1))

                       (TAKE-TRICK (TRICK-WINNER))
6:8                    --- [ELABORATE by RULE124] --->
                       (EACH C3 (CARDS-PLAYED)
                          (TAKE (TRICK-WINNER) C3))

               (DURING (EACH C3 (CARDS-PLAYED)
                          (TAKE (TRICK-WINNER) C3))
                       (FOR-SOME C1 (POINT-CARDS) (TAKE ME C1)))
6:9            --- [COLLECT by RULE58] --->
               (EXISTS C3 (CARDS-PLAYED)
                       (DURING (TAKE (TRICK-WINNER) C3)
                               (FOR-SOME C1 (POINT-CARDS)
                                  (TAKE ME C1))))
```

```
                         (DURING (TAKE (TRICK-WINNER) C3)
                                 (FOR-SOME C1 (POINT-CARDS)
                                    (TAKE ME C1)))
6:10                     --- [COLLECT by RULE100] --->
                         (EXISTS C1 (POINT-CARDS)
                                 (DURING (TAKE (TRICK-WINNER) C3)
                                         (TAKE ME C1)))

                                 (DURING (TAKE (TRICK-WINNER) C3)
                                         (TAKE ME C1))
6:11-22                          --- [by RULE43, simplification] --->
                 (AND [= (TRICK-WINNER) ME]
                      [TRICK-HAS-POINTS])

                         (TRICK-WINNER)
6:24                     --- [ELABORATE by RULE124] --->
                         (PLAYER-OF (WINNING-CARD))

6:25                     --- [ELABORATE by RULE124] --->
                         (THE P2 (PLAYERS)
                            (= (CARD-OF P2) (WINNING-CARD)))

                      (= (THE P2 (PLAYERS)
                            (= (CARD-OF P2) (WINNING-CARD)))
                         ME)
6:26-28               --- [REMOVE-QUANT by RULE131, analysis] --->
(ACHIEVE (NOT (AND [= (CARD-OF ME) (WINNING-CARD)]
                      [TRICK-HAS-POINTS])))
```

This expression is a predicate on the choice variable (CARD-OF ME), but is non-operational since it depends on the unpredictable (WINNING-CARD). The rest of the derivation consists of reformulating the expression to make it computable.

A similar reformulation occupies most of DERIV7, where "get the lead" is reformulated as "win the trick" and eventually as "play a high card".

4.3.1.3 Reformulating a Goal as an Executable Action

Another important problem that arises in forming a plan is reformulating an abstract event description as an executable action (§2.4). For instance, in DERIV8, much of the process of deriving a plan to flush the Queen consists of finding an action that will get rid of one of the spades held by whoever has the Queen:

```
            (REMOVE-1-FROM
               (SET-OF C3 (DIFF (CARDS) (SET QS)) (LEGAL (QSO) C3)))
8:41-61     --- [by analysis] --->
            (PLAY (QSO) C3)
```

The action found is for the player with the Queen to play some spade C3. The rest of the derivation, discussed elsewhere in more detail (§2.5.2), consists of figuring out how to force player (QSO) to play a spade.

A similar reformulation problem (§2.5.1) occurs in DERIV8 in the course of constructing a plan to get void in suit S0:

```
               (REMOVE-1-FROM
                  (SET-OF C1 (CARDS-IN-HAND ME) (IN-SUIT C1 S0)))
8:4-13         --- [by analysis] --->
               (AND [PLAY ME C1] [IN-SUIT C1 S0])
```

Here, the action that removes a card in suit S0 from player ME's hand is for player ME to play a card in suit S0.

4.3.2. Reformulating an Expression to Satisfy the Requirements of a Method

In order to apply an operationalization method to a problem, it is typically necessary to reformulate the problem in the form required by the method. This may involve rephrasing the overall problem to match the general problem statement for the problem, or reformulating a component of the problem in a special form dictated by the method.

4.3.2.1 Reformulating an Expression to Match a Method Problem Statement

The application of a rule expressing an operationalization method (§2) is often preceded by one or more steps that transform the original problem into a form that matches the rule (§1.6). For example, in DERIV4, the initial expression (QS-OUT) is reformulated in terms of set membership so as to match the pigeon-hole rule (§2.2):

```
         (QS-OUT)
4:1      --- [ELABORATE by RULE124] --->
         (OUT QS)

4:2      --- [ELABORATE by RULE124] --->
         (EXISTS P1 (OPPONENTS ME) (HAS P1 QS))

                 (HAS P1 QS)
4:3              --- [ELABORATE by RULE124] --->
                 (AT QS (HAND P1))

4:4              --- [ELABORATE by RULE124] --->
                 (= (LOC QS) (HAND P1))

         (EXISTS P1 (OPPONENTS ME) (= (LOC QS) (HAND P1)))
4:5      --- [COLLECT by RULE161] --->
         (EXISTS Y1 (PROJECT HAND (OPPONENTS ME)) (= (LOC QS) Y1))

4:6      --- [REMOVE-QUANT by RULE162] --->
         (IN (LOC QS) (PROJECT HAND (OPPONENTS ME)))
```

At this point it is possible to apply the pigeon-hole rule

RULE169: (in (f ...) S) -> (not (in (f ...) (diff (range f) S)))

In DERIV8, the expression (VOID PO SO) is reformulated in terms of empty sets:

```
          (ACHIEVE (VOID ME SO))
    8:1-2 --- [by analysis] --->
          (ACHIEVE (EMPTY (SET-OF C1 (CARDS-IN-HAND ME)
                                  (IN-SUIT C1 SO))))
```

This makes it possible to apply the set depletion rule (§2.5):

RULE6: (achieve (empty S)) -> (until (empty S) (achieve (remove-1-from S)))

In DERIV3, the goal (SUBSET (LEGALCARDS (QSO)) (SET QS)) is mapped onto RULE6:

```
          (ACHIEVE (SUBSET (LEGALCARDS (QSO)) (SET QS)))
    3:1-2 --- [by analysis] --->
          (ACHIEVE (EMPTY (DIFF (LEGALCARDS (QSO)) (SET QS))))
```

In DERIV2, the goal (AVOID-TAKING-POINTS (TRICK)) is reformulated as a predicate on a scenario so as to match the problem statement for HSM (§3.3):

```
    (AVOID-TAKING-POINTS (TRICK))
    2:1-2 --- [by analysis] --->
    (ACHIEVE (NOT (DURING (TRICK) (TAKE-POINTS ME))))
```

Note that the partial-matching strategy described earlier for detecting potential applicability of methods (§4.2.2.1) would not work very well in any of the above examples. In most of them, the initial expression (f ...) requires top-level reformulation to fit the method. The only exceptions above involve reformulating expressions of the form (achieve P) to fit RULE6, but partial matches such as the one between (ACHIEVE (VOID ME SO)) and (achieve (empty S)) are very weak clues since any goal (achieve P) would match just as well. The quality of the match might be enhanced by introducing suitable features like "set-related predicate" as a common generalization of VOID and EMPTY. The match between these two terms would be considered better than a match between two totally unrelated terms. This would help distinguish good (highly predictive) matches from bad ones. The intersection search strategy also appears useful in the examples above.

4.3.2.2 Reformulating an Expression to Provide Information for a Method

An expression may be in the overall form required by an operationalization rule, but have a component in the wrong form. For example, consider the intersection search rule

RULE57: (during s e) -> nil if s and e have no common sub-events

RULE57 requires that s and e be expressed in terms of known actions like GET-CARD, rather than specifications of their effects like "cause a card to become out" or 'undo a void". This requirement motivates the following reformulation of the abstract event "card X1 is caused to be out" as "some opponent P1 gets card X1":

```
(DURING [ROUND-IN-PROGRESS]
        [CAUSE (OUT X1)])

10:5    --- [ELABORATE by RULE124] --->
        [CAUSE (EXISTS P1 (OPPONENTS ME) (HAS P1 X1))]

10:6    --- [by RULE329] --->
        [EXISTS P1 (OPPONENTS ME) (CAUSE (HAS P1 X1))]

                                         (HAS P1 X1)
10:7                                     --- [by RULE124] --->
                                         (AT X1 (HAND P1))

                                (CAUSE (AT X1 (HAND P1)))
10:8                            --- [RECOGNIZE by RULE358] --->
                                (MOVE X1 LOC1 (HAND P1))

10:9                            --- [RECOGNIZE by RULE123] --->
(DURING [ROUND-IN-PROGRESS]
        [EXISTS P1 (OPPONENTS ME) (GET-CARD P1 X1 LOC1)])
```

The reformulated expression can now be used as the value of e in RULE57, since it is expressed in terms of the known action GET-CARD. Similarly, in DERIV12, the abstract event "player P0 ceases to be void in suit S0" is reformulated as "P0 gets a card in suit S0":

```
(DURING [ROUND-IN-PROGRESS]
        [UNDO (VOID P0 S0)])
12:2            --- [ELABORATE by RULE124] --->
        [UNDO (NOT (EXISTS C1 (CARDS-IN-HAND P0)
                            (IN-SUIT C1 S0)))]
12:3    --- [RECOGNIZE by RULE377] --->
        [CAUSE (EXISTS C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0))]

                            (CARDS-IN-HAND P0)
12:4                        --- [ELABORATE by RULE124] --->
                            (SET-OF C2 (CARDS) (HAS P0 C2))

                (EXISTS C1 (SET-OF C2 (CARDS) (HAS P0 C2))
                    (IN-SUIT C1 S0))
12:5            --- [by RULE135] --->
        [CAUSE (EXISTS C1 (CARDS) (AND [HAS P0 C1]
                                       [IN-SUIT C1 S0]))]
12:6    --- [by RULE329] --->
        [EXISTS C1 (CARDS) (CAUSE (AND [HAS P0 C1]
                                       [IN-SUIT C1 S0]))]

                             (CAUSE (AND [HAS P0 C1]
                                         [IN-SUIT C1 S0]))
12:7                         --- [REDUCE by RULE35] --->
                             (AND [CAUSE (HAS P0 C1)]
                                  [IN-SUIT C1 S0])
```

```
                                                   (HAS P0 C1)
12:8                                               --- [by RULE124] --->
                                                   (AT C1 (HAND P0))

                                          (CAUSE (AT C1 (HAND P0)))
12:9                                      --- [RECOGNIZE by RULE358] --->
                                          (MOVE C1 LOC1 (HAND P0))
12:10                                     --- [RECOGNIZE by RULE123] --->
(DURING [ROUND-IN-PROGRESS]
        [EXISTS C1 (CARDS) (AND [GET-CARD P0 C1 LOC1]
                                [IN-SUIT C1 S0]))])
```

As before, the reformulated expression can subsequently be used as the value of e in RULE57, since it is expressed in terms of the known action GET-CARD.

4.4. Reformulation: Summary

A variety of reformulation problems appear in the derivations. Some are represented explicitly, while others are implicit in the directionality of a derivation. This is not an inherent distinction between two kinds of reformulation problems, but rather an artifact of FOO's particular set of rules and the absence of a problem-solving component: explicit representation of a reformulation problem requires some target description of what the initial expression is to be reformulated as, and FOO lacks the sort of means-end analysis that might propose such targets (§8.1.1). Exceptions -- explicit reformulation problems -- occur when the decision to apply a particular strategy, encoded in a rule, entails solving a particular reformulation problem. For example, the decision to use HSM to evaluate a predicate generates the subproblem of reformulating the predicate as a function of the choice sequence (§3.3.3).

Reformulation problems implicit in the existing derivations could be explicitly generated by encoding the underlying strategies as rules:

Reformulate a goal as a computable predicate on a choice variable.

Reformulate an expression in terms of an observable action or condition.

Adding such rules to FOO would involve enriching the rule language to include concepts like "choice variable" and "observable" and supplying procedures to identify instances of them, e.g., find all the choice variables in a given task description. This is closely related to the problem of representing the run-time capabilities of the task agent (§8.1.1).

Since the implicit/explicit distinction depends on what happens to be implemented in FOO, rather than on inherent problem features, it provides a rather uniformative criterion for classifying reformulation problems. Better criteria include:

Target description: reformulate expression e

in terms of concept f

as a function of quantity e'

Accuracy: reformulated expression

is semantically equivalent to the original expression

approximates the original expression

Purpose: transform the expression to fit

the run-time capabilities of the task agent

the problem statement of a method

These criteria provide a taxonomy of reformulation problems:

1. Reformulate expression e in terms of concept f: solve e = $h(f(x_1, ..., x_n))$ for $h, x_1, ..., x_n$
   a. Fit the runtime capabilities of the task agent
      i. Reformulate e in terms of an observable entity
         1. Reformulate e in terms of an observable condition
         2. Reformulate e in terms of an observable action
         3. Reformulate e in terms of an object's observable state
   b. Reformulate e in terms of an executable action
   c. Fit the requirements of a method
      i. Reformulate e to match the problem statement of a method
      ii. Reformulate an argument of e to satisfy a condition of the method
2. Reparameterize expression e as a function of quantity e': solve e = h(e') for h
   a. Fit the runtime capabilities of the task agent
      i. Reformulate a goal as a function of a choice variable
3. Approximate expression e as a 0-th order function of quantity v: (function-of v (D_v e))

Chapter 5
Analysis Methods

The preceding chapters described various methods for operationalizing expressions. The rules expressing these methods are "high-level" in that they make operationalization decisions but do not carry out the work of implementing them. In particular, they do not tell how to:

1. Reformulate the original problem to fit the rule.
2. Verify the rule condition.
3. Convert the result of applying the rule into a useful form.

The broad term "analysis" includes all three of these processes, which account for the bulk of the derivations and the majority of the rules in FOO. This chapter describes the methods used to perform such analysis:

1. Simplification (§5.2)
2. Partial matching (§5.3)
3. Techniques for propagating assumptions, solving for variables, and finding examples (§5.4)
4. Expansion of recursive definitions (§5.5)
5. Intersection search (§5.6)
6. Problem separation (§5.7)
7. Translation (§5.8)
8. Problem reduction (§5.9)
9. Problem restructuring (§5.10)

5.1. Taxonomy of Analysis Methods

These methods fit into a rough taxonomy based on how they are used, *i.e.*, the syntactic and semantic effects of applying them:

1. Methods that preserve semantic equivalence
    a. Methods always safe to use (so never require backtracking) (§5.2)
        i. Simplification to a constant (§5.2)
            1. Systematic evaluation (§5.2.2)
            2. Opportunistic evaluation (§5.2.3)
        ii. Simplification to a non-constant (§5.2.4)
    b. Heuristic methods (not always useful even though valid)
        i. Translation (§5.8)
            1. Elaboration (§5.8.2)
                a. Expand recursive definition (§5.5)
            2. Recognition (§5.8.3)
            3. Rephrasing (§5.8.4)
        ii. Intersection search (§5.6)
2. Methods that sometimes violate semantic equivalence
    a. Problem reduction (§5.9)
        i. Partial matching (§5.3)
            1. Solve recurrence (§5.3.5)
        ii. Find example (§5.4.4)
    b. Restructuring (§5.10) (transpose (§5.10.1), transfer (§5.10.2), functionalize (§5.10.3))
        i. Problem separation (§5.7) (split (§5.7.4) and propagate (§5.7.5))
            1. Case analysis (§5.7.1)
            2. Condition factoring (§5.7.2)
            3. Conjunctive subgoaling (§5.7.3)
        ii. Merge problems (§5.7.6)

c. Propagate information (§5.4)

i. Use assumption (§5.4.3)

ii. Bind variable (§5.4.1)

iii. Restrict variable (§5.4.2)

5.2. Simplification

The purpose of simplification is to convert an expression into a quasi-canonical form to enable the subsequent application of operationalization or analysis methods. Simplification rules are characterized by the following properties:

1. The new expression is semantically equivalent to the old one, *i.e.*, has the same denotation.
2. The new expression is simpler according to some criterion of relative complexity.
3. The new expression can be operationalized in no more steps than the old one.

Property (1) excludes transformation rules that sometimes produce an expression whose meaning differs from the original expression, such as

RULE43: (R [f e_1 ... e_n] [f e_1' ... e_n']) -> (and [= e_1 e_1'] ... [= e_n e_n']) if R is reflexive

The complexity criterion of property (2) may depend not only on the size of an expression but on the number of function calls in it. For example, the transformation from the intensional expression `(SET-OF I (LB:UB 1 100) (IS-PRIME I))` to its extension `(SET 2 3 5 ... 97)` could be considered simplification, even though the latter expression contains more symbols.

Property (3) says that simplification is always a good idea, or at least never hurts. This property is *extrinsic*, since it depends not only on the old and new expressions but on the set of operationalization and analysis methods available. Essentially, every useful method that applies to the original expression should apply to the simplified one as well (except perhaps for the methods used to make the simplification itself). Thus not every transformation of an expression into a shorter semantic equivalent constitutes simplification. For example, recognizing an expression as an instance of a known concept is not simplification, since useful transformations that apply to the expanded form of the expression may not apply to the compact form.

An example of simplification is deleting a null clause from a disjunction:

(or P_1 ... nil ... P_n) -> (or P_1 ... P_n)

This may be a waste of time -- for instance, if P_1 is known to be true, it just delays the transformation

(or T P_2 ... nil ... P_n) -> T

However, deleting the null clause is harmless in that any useful transformations that can be applied to (or P_1 ... nil ... P_n) can still be applied to (or P_1 ... P_n), in particular the transformation

(or T P_2 ... P_n) -> T

It is convenient to distinguish between three kinds of simplification rules:

1. Systematic evaluation (§5.2.2)
2. Opportunistic evaluation (§5.2.3)
3. Simplification to a non-constant (§5.2.4)

These three kinds of simplification rules are distinguished as follows. *Evaluation* transforms an expression into a constant. *Systematic evaluation* of an expression (f e_1 ... e_n) is performed by a *uniform procedure* for f that computes the value of (f e_1 ... e_n) given the values of e_1, ..., e_n, *i.e.*, evaluates any expression (f c_1 ... c_n) where c_1, ..., c_n are constants. In contrast, *opportunistic evaluation* computes the value of (f e_1 ... e_n) by means of a special-case rule that allows non-constants as e_1, ..., e_n but only works for expressions that fit the rule. Finally, *simplification to a non-constant* reduces (f e_1 ... e_n) to a simpler expression without computing its value. (§5.2.1) illustrates these classes by example. (§5.2.2), (§5.2.3), and (§5.2.4) list the rules in each class, and (§5.2.6) summarizes the differences between them.

5.2.1. An Illustrative Example of Simplification

An example drawn from DERIV3 illustrates the concepts of *systematic evaluation*, *opportunistic evaluation*, and *simplification to a non-constant*, and their role in operationalization. Prior to the point where the example begins, the initial goal ("flush out the Queen of spades") has been split into two subgoals:

```
(ACHIEVE (FLUSH QS))
3:1-12   --- [by analysis] --->
(ACHIEVE (AND [SUBSET (LEGALCARDS (QSO)) (SET QS)]
              [SUBSET (SET QS) (LEGALCARDS (QSO))]))
```

The first subgoal means "make player (QSO) have no legal cards other than the Queen of spades." The second subgoal reduces to "make it legal for the player (QSO) who has the queen of spades to play it":

```
3:14-19         --- [by analysis] --->
                (LEGAL (QSO) QS)
3:20            --- [ELABORATE by RULE124] --->
                (AND [HAS (QSO) QS]
                     [=> (LEADING (QSO))
                         (OR [CAN-LEAD-HEARTS (QSO)]
                             [NEQ (SUIT-OF QS) H])]
                     [=> (FOLLOWING (QSO))
                         (OR [VOID (QSO) (SUIT-LED)]
                             [IN-SUIT QS (SUIT-LED)])])
```

The first of the three conjuncts reduces to true by definition of (QSO); the third is satisfied by assuming that spades are led. The treatment of the second conjunct illustrates the role of the transformations discussed in this section. The point of the example is that this expression can't be evaluated *systematically* without knowing whether it's legal for player (QSO) to lead hearts, but it can still be evaluated *opportunistically*, and doing so ultimately enables the top-level goal to be *simplified* to the point where a high-level strategy can be applied.

First (SUIT-OF QS) is evaluated as S (spades) by a "fact rule" (RULE2) that simulates a procedural definition of SUIT-OF or a general mechanism for retrieving properties of multi-featured objects (in this case the suit of a card):

```
                                  (SUIT-OF QS)
3:30                              --- [EVAL by RULE2] --->
                                  S
```

An expression with a known function and constant arguments can be *systematically evaluated:*

```
                             (NEQ S H)
3:31                         --- [COMPUTE by RULE236] --->
                             T
```

A disjunction containing a term known to be true can be *opportunistically evaluated* as true without knowing the values of the other terms:

```
                           (OR [CAN-LEAD-HEARTS (QSO)] T)
3:32                       --- [EVAL by RULE184] --->
                           T
```

Similarly, an implication whose consequent is true can be opportunistically evaluated as true regardless of the antecedent:

```
                         (=> (LEADING (QSO)) T)
3:33                     --- [EVAL by RULE47] --->
                         T
```

An alternative approach is to assume that player (QSO) won't be leading, and use the opportunistic transformation

(=> nil P) -> T

The subgoal of making it legal for player (QSO) to play the Queen has now been reduced to true (assuming that spades will be led). This permits the overall goal to be simplified:

```
(ACHIEVE (AND [SUBSET (LEGALCARDS (QSO)) (SET QS)] T))
3:36      --- [SIMPLIFY by RULE343] --->
(ACHIEVE (SUBSET (LEGALCARDS (QSO)) (SET QS)))
```

The simplified goal can be reformulated based on a set identity:

```
3:37      --- [by RULE4] --->
(ACHIEVE (EMPTY (DIFF (LEGALCARDS (QSO)) (SET QS))))
```

The reformulated goal matches the left side of the set-depletion rule (§2.5):

RULE6: (achieve (empty S)) -> (until P (achieve (remove-1-from S))),
where P is the original goal

To make a set empty, remove its elements one at a time.

The simple strategy expressed by this rule provides the basis for the plan subsequently derived (§2.5.1). Note that the simplification at step 3:36 puts the overall goal into a quasi-canonical form that allows this strategy to be applied.

The moral of this story is three-fold:

1. Systematic evaluation is useful when applicable.
2. Opportunistic evaluation sometimes succeeds where systematic evaluation would either fail or require more work.
3. Simplification helps reduce expressions to a quasi-canonical form to which higher-level transformations can be applied.

The rest of this section presents FOO's rules for evaluation and simplification.

5.2.2. Systematic Evaluation

Clearly, systematic evaluation is the most powerful type of analysis -- when it is possible. FOO's rule for systematic evaluation is

RULE236: $(f\,c_1 \ldots c_n)$ -> c,
where c is the result of applying the evaluation procedure for f to the constants $c_1 \ldots c_n$

The following examples illustrate RULE236's use:

```
(AND T T)
2:51 --- [EVAL by RULE236] --->
T

(NEQ S H)
3:31 --- [EVAL by RULE236] --->
T

(AND T (AND T) T)
3:57   --- [EVAL by RULE236] --->
(AND T T T)
3:58 --- [EVAL by RULE236] --->
T

(NOT T)
3:76 --- [EVAL by RULE236] --->
NIL

(NOT NIL)
3:85 --- [EVAL by RULE236] --->
T

(D- NIL)
9:52 --- [EVAL by RULE236] --->
NIL

(D+ NIL (D- INCREASING))
9:58    --- [EVAL by RULE236] --->
(D+ NIL DECREASING)
9:59 --- [EVAL by RULE236] --->
DECREASING

(- 2 1)
13:34 --- [EVAL by RULE236] --->
T
```

There are many other places in the derivations where an expression (f e_1 ... e_n) was transformed into a constant, but RULE236 was not used because:

1. It was not possible to evaluate one or more of the arguments e_i;
2. It was not necessary to evaluate all the arguments; or
3. FOO lacked a general evaluation procedure for the function f.

5.2.3. Opportunistic Evaluation

Special-case *opportunistic* rules for evaluating an expression (f e_1 ... e_n) are useful when

1. It is not possible to evaluate one or more of the arguments e_i;
2. The expression can be evaluated without evaluating all the arguments; or
3. There is no general evaluation procedure available for the function f.

FOO's opportunistic evaluation rules are listed below along with some of the expressions they were used to evaluate. For brevity, only the step number is shown; the computed value of the expression is given by the right-hand side of the rule. (The rule index lists all uses of each rule.) The following list omits certain rules that produce a constant but are more appropriately discussed elsewhere, *e.g.*, under intersection search (§5.6).

RULE47: (=> P T) -> T

```
3:28 (=> (FOLLOWING (QSO)) T)
3:33 (=> (LEADING (QSO)) T)
```

RULE179: (R e e) -> T if R is reflexive

```
3:73 (= S S)
5:33 (= (SUIT-OF (CARD-OF ME)) (SUIT-OF (CARD-OF ME)))
9:34 (= X2 X2)
14:80 (= (NEXT NOTE) (NEXT NOTE))
```

RULE184: (or ... T ...) -> T

```
3:27 (OR [VOID (QSO) (SUIT-LED)] T)
3:32 (OR [CAN-LEAD-HEARTS (QSO)] T)
4:25 (OR T
         [IN (HAND ME) (PILES (PLAYERS))]
         [IN (HAND ME) (SET DECK POT HOLE)])
```

RULE288: (# (collection e_1 ... e_n)) -> n

```
14:24 (# (SET (NEXT NOTE)))
```

RULE289: (# nil) -> 0

```
14:23 (# NIL)
```

RULE293: (in (one-of S) S) -> T

```
14:33 (IN (HIGHEST CANTUS-1) CANTUS-1)
```

RULE337: (subset (prefix S i) S) -> T

```
2:98 (SUBSET (PREFIX CARDS-PLAYED1 ANY) CARDS-PLAYED1)
```

5.2.4. Simplification to a Non-constant

Sometimes an expression can be simplified but not evaluated. By reducing an expression to a quasi-canonical form, such simplification may enable (and at worst does not prevent) the application of useful rules. FOO's rules for this kind of simplification appear below, together with some of the expressions they were used to simplify. For brevity, the resulting expression is only shown for the first example of each rule. The examples are included to clarify each rule's operation and the variety of expressions to which it applies, but are not very interesting reading.

RULE11: (and P_1 ... T ... P_n) -> (and P_1 ... P_n)

```
2:33-34 (AND [IN P4 (OPPONENTS ME)] T T)
   ---> (AND [IN P4 (OPPONENTS ME)])

2:53 (AND T [IN-SUIT C11 (SUIT-LED)])

2:60 (AND T [LAMBDA (P1 C21) ...])

3:82 (AND [PLAY (QSO) C3] T [=> (FOLLOWING (QSO)) ...] [IN C3 ...])
```

RULE12: (=> T P) -> P

```
3:86 (=> T (IN-SUIT C3 S)) ---> (IN-SUIT C3 S)

13:26 (=> T (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
               (USABLE-INTERVAL! INTERVAL1)))

13:50 (=> T (=< (RANGE PROJECT1) (MAJOR 10)))
```

RULE19: (diff (join S_1 ... S ... S_n) S) -> (join S_1 ... S_n) if S_1, ..., S, ..., S_n are mutually disjoint

```
4:16 (DIFF (UNION [HANDS (PLAYERS)]
                  [PILES (PLAYERS)]
                  [SET DECK POT HOLE])
           (HANDS (PLAYERS)))
---> (UNION [PILES (PLAYERS)] [SET DECK POT HOLE])
```

RULE88: (not (not P)) -> P

```
2:43 (NOT (NOT (EXISTS C1 (CARDS-IN-HAND P5 (IN-SUIT C1 (SUIT-LED))))))
---> (EXISTS C1 (CARDS-IN-HAND P5 (IN-SUIT C1 (SUIT-LED))))

5:38 (NOT (NOT (HIGHER (FIND-ELT (CARDS-IN-SUIT-LED (CARDS-PLAYED)))
                       (CARD-OF ME))))
```

RULE108: (Q x S (and ... P ...)) -> (and P [Q x S (and ...)]) if P is independent of x, assuming S is non-empty

```
6:13 (EXISTS C3 (CARDS-PLAYED) (AND [IN C3 (POINT-CARDS)]
                                    [= (TRICK-WINNER) ME]))
---> (AND [= (TRICK-WINNER) ME]
          [EXISTS C3 (AND [IN C3 (POINT-CARDS)])])
```

RULE155: (Q x S P) -> P if P is independent of x, assuming S is non-empty

```
2:56 (EXISTS C11 (CARDS) (IN-SUIT C2 (SUIT-LED)))
---> (IN-SUIT C2 (SUIT-LED))

7:7 (FORALL X1 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
       (NOT (LOW (CARD-OF ME))))
```

RULE176: (join e_1 ... nil ... e_n) -> (join e_1 ... e_n)

```
2:8 (APPEND (CHOICE-SEQ-OF (EACH P1 (PLAYERS) (PLAY-CARD P1)))
            NIL)
--> (APPEND (CHOICE-SEQ-OF (EACH P1 (PLAYERS) (PLAY-CARD P1))))

3:78 (OR NIL [IN-SUIT C3 S])
```

```
4:39 (OR NIL [AT QS POT]
             [AT QS HOLE]
             [AT QS (HAND ME)]
             [IN (LOC QS) (PILES (PLAYERS))])

4:49 (OR [WAS-DURING (CURRENT ROUND-IN-PROGRESS)
                (CAUSE (AT QS (PILE P3)))]
         NIL)
```

RULE177: $(C\ e_1 \dots (C\ e_1' \dots e_k') \dots e_n) \rightarrow (C\ e_1 \dots e_n\ e_1' \dots e_k')$,
where C is a commutative operator (*e.g.*, AND, OR, +, UNION, etc.)

```
3:45 (AND [AND [UNDO (HAS (QSO) C3)]
               [=> (LEADING (QSO)) ...]
               [=> (FOLLOWING (QSO)) ...]]
          [IN C3 (DIFF (CARDS) (SET QS))])
---> (AND [UNDO (HAS (QSO) C3)]
          [=> (LEADING (QSO)) ...]
          [=> (FOLLOWING (QSO)) ...]
          [IN C3 (DIFF (CARDS) (SET QS))])
4:28 (UNION [UNION [PILES (PLAYERS)] [SET DECK POT HOLE]]
            [SET (HAND ME)])

4:33 (OR [OR [= (LOC QS) DECK] [= (LOC QS) POT] [= (LOC QS) HOLE] ...]
         [IN (LOC QS) (PILES (PLAYERS))])

14:38 (AND [NOT (CHANGE (CLIMAX CANTUS-1))]
           [AND [= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1]
                [NOT (= (NEXT NOTE) (CLIMAX CANTUS-1))]]])
```

RULE178: (C e) -> e, where C is a commutative operator

```
2:9 (APPEND [CHOICE-SEQ-OF (EACH P1 (PLAYERS) (PLAY-CARD P1))])
--> (CHOICE-SEQ-OF (EACH P1 (PLAYERS) (PLAY-CARD P1)))

2:35 (AND [IN P4 (OPPONENTS ME)])

2:61 (AND [LAMBDA (CARDS-PLAYED1) ...])

3:79 (OR [IN-SUIT C3 S])

9:46 (D+ [D* (DEPENDENCE ...) (DEPENDENCE ...)])

14:65 (UNION [IF (= ...) (SET (NEXT NOTE)) NIL])
```

RULE185: (if T x y) -> x

```
4:26 (IF T (SET (HAND ME)) NIL)
---> (SET (HAND ME))

14:67 (IF T (SET (NEXT NOTE)) NIL)
```

RULE188: (C ... e ... e ...) -> (C ... e ...), *i.e.*, remove duplicate occurrence of e,
where C is a commutative operator

```
2:70 (AND [IN ME (INDICES-OF CARDS-PLAYED1)]
          [IN ME (INDICES-OF CARDS-PLAYED1)])
---> (AND [IN ME (INDICES-OF CARDS-PLAYED1)])
```

```
13:90 (AND [IN CLIMAX1 PROJECT1]
           [=< (#OCCURRENCES CLIMAX1 PROJECT1) 1]
           [IN CLIMAX1 PROJECT1]
           [FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1))]
           [FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
              (USABLE-INTERVAL! INTERVAL1)]
           [=< (RANGE PROJECT1) (MAJOR 10)])
```

RULE216: (f([inverse f] e)) -> e

```
9:19 (NOT ([INVERSE NOT] (DISJOINT (CARDS-IN-HAND P0) S1)))
---> (DISJOINT (CARDS-IN-HAND P0) S1)
```

RULE253: (lb:ub k k) -> {k}

```
13:35 (LB:UB 1 1) ---> (SET 1)
```

RULE255: (Q x (collection e) P_x) -> P_e

```
2:104 (EXISTS C23 (LIST C21) (HAS-POINTS C23)) ---> (HAS-POINTS C21)
```

RULE281: (one-of (collection e)) -> e

```
14:45 (HIGHEST (LIST (NEXT NOTE))) ---> (NEXT NOTE)
```

RULE310: (apply append (each x S (list e_x))) -> (each x S E_x)

```
2:13 (APPLY APPEND (EACH P1 (PLAYERS)
                         (LIST (CHOOSE (CARD-OF P1) (LEGALCARDS P1)
                                  (PLAY P1 (CARD-OF P1))))))
---> (EACH P1 (PLAYERS) (CHOOSE (CARD-OF P1) (LEGALCARDS P1)
                              (PLAY P1 (CARD-OF P1))))

13:7 (APPLY APPEND (EACH I1 (LB:UB 1 (CANTUS-LENGTH))
                         (LIST (CHOOSE (NOTE I1) (TONES) (NOTE I1)))))
```

RULE341: (C i e) -> e, where C is a commutative operator and i is its identity element

```
3:19 (AND T [LEGAL (QSO) QS]) ---> (LEGAL (QSO) QS)

5:9 (OR NIL [DURING (TAKE-TRICK (TRICK-WINNER)) (TAKE ME QS)])

9:48 (D* INCREASING [DEPENDENCE (PR-DISJOINT-FORMULA #H #S #U) #S])

13:40 (AND T [LAMBDA (I1 NOTE1) ...])
```

RULE343: (C e i) -> e, where C is a commutative operator and i is its identity element

```
3:36 (AND [SUBSET (LEGALCARDS (QSO)) (SET QS)] T)
---> (SUBSET (LEGALCARDS (QSO)) (SET QS))

9:53 (D+ [DEPENDENCE (#CHOOSE (- #U #S) #H) #S] NIL)

14:26 (+ [#OCCURENCES (CLIMAX CANTUS-1) CANTUS-1] 0)
```

5.2.5. Automated Simplification

Simplification does not hinder the subsequent application of other methods, and often helps by converting expressions into a quasi-canonical form that fits an available method. FOO's simplification rules could therefore safely be incorporated in a procedure automatically applied after each step in a derivation. This would change the basic problem-solving cycle from *select-a-rule-and-apply-it* (analogous to the "recognize-act" cycle of production systems [Newell 72]) to *select-apply-simplify*. The latter architecture is exemplified by programs for symbolic integration [Moses 71] [Slagle 63] and other kinds of manipulation.

Using this approach in an operationalization problem-solver would decrease derivation length, *i.e.*, the number of operators needed to transform a problem into an operational solution. Thus the depth of the search space would be reduced. However, the decision whether to incorporate a given transformation rule in the canonicalization procedure would have to be made very carefully, since it depends not only on the rule itself but also on its potential interference with other methods. Incorrectly classifying a rule as a simplification rule could make it impossible to generate a non-canonical form required by another rule. For example, consider

RULE213: ([lambda (x_1 ... x_n) e] e_1 ... e_n) -> e',
where e' is the result of substituting e_1 ... e_n for x_1 ... x_n throughout e

RULE213 is useful in DERIV14:

```
([LAMBDA (J) (NOTE (+ J 1))] T1)
14:86 --- [SIMPLIFY by RULE213] --->
(NOTE (+ T1 1))
```

However, RULE213 can interfere with a rule for instantiating the HSM schema (§3.3.3):

RULE317: (HSM with (reformulated-problem : ([lambda (s) P_s] choices)))
-> (HSM with (solution-test : (lambda (s) P_s)) ...)

Specifically, applying RULE213 whenever possible would automatically reduce any expression of the form ([lambda (s) P_s] choices) to $P_{choices}$, preventing the application of RULE317.

5.2.6. Simplification: Summary

Simplification transforms an expression into a simpler but semantically equivalent one. FOO has three kinds of simplification rules:

1. *Systematic evaluation* transforms an expression ($f\ c_1 \ldots c_n$) into a constant by applying a uniform evaluation procedure for f to the constant arguments $c_1, \ldots, c_n$.
2. *Opportunistic evaluation* uses a special-case rule to transform an expression into a constant without evaluating its arguments.
3. *Simplification to a non-constant* transforms an expression into a quasi-canonical form to which additional methods may apply.

Simplification preserves semantic equivalence and typically requires little computation. It does not hinder the subsequent application of other methods, and often helps by converting expressions into a quasi-canonical form that fits an available method. FOO's simplification rules could therefore safely be incorporated in a canonicalization procedure automatically applied after each step in a derivation, although the decision to classify a rule as a simplification rule -- *i.e.*, to apply it whenever possible -- would depend not only on the rule itself but also on the other rules available, so as not to interfere with them. Incorrectly classifying a rule as a simplification rule could interfere with the transformation of expressions into a non-canonical form required by some other rule.

5.3. Partial Matching

FOO uses *partial matching* [Hayes-Roth 78b] to simplify a relation between similar expressions by equating their corresponding arguments. Unlike the methods described in (§5.2), partial matching is generally *sound* but not always *valid*. It can be used to make *plausible inferences* that are logically unjustified but empirically true, or at least lead to good task performance. Partial matching permits analysis of an expression without expanding the definitions of all the terms in it. This is especially useful if the expression contains terms whose definitions are unknown.

The rest of this section is organized as follows. (§5.3.1) describes the typical scenario surrounding the use of partial matching. (§5.3.2) illustrates this scenario by means of an example. (§5.3.3) discusses the logical status of the partial matching transformation. (§5.3.4) presents FOO's rules for partial matching. (§5.3.5) describes the use of partial matching to induce function definitions from recurrence equations. Finally, (§5.3.6) summarizes the purpose, use, and result of partial matching.

5.3.1. A General Scenario for Partial Matching

Partial matching in FOO is typically used to reduce a proposition (R [f e_1 ... e_n] [f e_1' ... e_n']), where R is a reflexive binary relation, to the assertion that e_i = e_i' for each i = 1, ..., n. It usually occurs as part of the following *partial matching scenario*:

1. *Reformulate* an expression in terms of a relation between two entities of the same type.
2. *Translate* the descriptions of these entities into common terms.
3. *Restructure* the expression to make the arguments of the relation have the same form.
4. *Partial-match* the arguments by equating their corresponding components.
5. *Simplify* the resulting conjunction of equalities in its surrounding context.

The first three steps satisfy requirements of the partial matching method:

Reformulating the initial expression provides the relation R.

Translating its arguments into common terms provides the function f.

Restructuring the expression provides the required form (R [f ...] [f ...]).

The fourth step makes the expression more nearly operational:

Partial-matching the arguments eliminates the possibly troublesome R and f.

The last step satisfies requirements of subsequently applied methods:

Simplifying the expression puts it in a quasi-canonical form.

When one of these requirements is satisfied serendipitously, the corresponding step is absent.

5.3.2. An Example of Partial Matching

An excerpt from DERIV6 illustrates the usefulness of partial matching. To operationalize the advice "avoid taking points during the trick," the definition of a trick is analyzed to determine the circumstances under which player ME takes points:

```
(AVOID (TAKE-POINTS ME) (TRICK))
6:2-6 --- [by analysis] --->
(ACHIEVE (NOT (DURING [TAKE-TRICK (TRICK-WINNER)]
                      [TAKE-POINTS ME])))
```

In other words, to avoid taking points, make sure you don't take any point cards when the winner takes the trick. In terms of the partial-matching scenario, this *reformulates* the problem in terms of the relation DURING between the event descriptions (TAKE-TRICK (TRICK-WINNER)) and (TAKE-POINTS ME), which are next *translated* into common terms:

```
                        (DURING [TAKE-TRICK (TRICK-WINNER)]
                                [TAKE-POINTS ME])
6:7-8                           --- [ELABORATE by RULE124] --->
                        (DURING [EACH C3 (CARDS-PLAYED)
                                   (TAKE (TRICK-WINNER) C3)]
                                [FOR-SOME C1 (POINT-CARDS)
                                   (TAKE ME C1)]))
```

In this case, the "common term" is the action TAKE. The next step *restructures* the expression into the form (DURING [TAKE ...] [TAKE ...]) by moving the quantifiers outside:

```
                        (DURING [EACH C3 (CARDS-PLAYED)
                                   (TAKE (TRICK-WINNER) C3)]
                                [FOR-SOME C1 (POINT-CARDS)
                                   (TAKE ME C1)])
6:9-10                  --- [by RULE58, RULE100] --->
                        (EXISTS C3 (CARDS-PLAYED)
                           (EXISTS C1 (POINT-CARDS)
                              (DURING [TAKE (TRICK-WINNER) C3]
                                      [TAKE ME C1])))
```

Now that both arguments to DURING have the form (TAKE <player> <card>), they can be *partial-matched* by equating their corresponding components:

```
                              (DURING [TAKE (TRICK-WINNER) C3]
                                      [TAKE ME C1])
6:11                          --- [by RULE43] --->
                              (AND [= (TRICK-WINNER) ME] [= C3 C1])
```

This transformation eliminates the predicate DURING, for which FOO has no general definition, and the action TAKE, whose details are irrelevant to the problem of avoiding points. The resulting expression is *simplified* by using the equality (= C3 C1) to eliminate the quantifier variable C1:

```
                        (EXISTS C3 (CARDS-PLAYED)
                           (EXISTS C1 (POINT-CARDS)
                              (AND [= (TRICK-WINNER) ME] [= C3 C1])))
6:12                       --- [REMOVE-QUANT by RULE59] --->
                        (EXISTS C3 (CARDS-PLAYED)
                           (AND [IN C3 (POINT-CARDS)]
                                [= (TRICK-WINNER) ME]))
```

The conjunct (= (TRICK-WINNER) ME) can be moved outside the scope of quantification:

```
                   (EXISTS C3 (CARDS-PLAYED)
                      (AND [IN C3 (POINT-CARDS)]
                           [= (TRICK-WINNER) ME]))
6:13-14            --- [SIMPLIFY-QUANT by RULE108, RULE178] --->
                   (AND [= (TRICK-WINNER) ME]
                        [EXISTS C3 (CARDS-PLAYED)
                           (IN C3 (POINT-CARDS))])
```

Further analysis leads to an operational solution:

```
(ACHIEVE (NOT (AND [= (TRICK-WINNER) ME]
                   [EXISTS C3 (CARDS-PLAYED)
                      (IN C3 (POINT-CARDS))])
6:15-43  --- [by analysis] --->
(ACHIEVE (=> (AND [IN-SUIT-LED (CARD-OF ME)] [TRICK-HAS-POINTS])
             (LOW (CARD-OF ME))))
```

That is, if you're following suit (or leading) in a trick liable to have points, play a low card to avoid taking points.

5.3.3. Logical Status of Partial Matching

The key transformation in the example was performed by a general rule for partial matching:

RULE43: (R [f e_1 ... e_n] [f e_1' ... e_n']) -> (and [= e_1 e_1'] ... [= e_n e_n']) if R is reflexive

RULE43 is obviously a logically *sufficient* transformation, *i.e.*,

e_i = e_i' for i = 1, ..., n *implies* (R [f e_1 ... e_n] [f e_1' ... e_n']) if R is reflexive

But when is the converse true, *i.e.*, when does RULE43 preserve logical equivalence? It does in the example: (during $event_1$ $event_2$) implies $event_1$ = $event_2$ if the two events are instances of the same action (in this case TAKE), since:

1. The simple model of the Hearts world assumes that primitive actions (like moving a card) occur in sequence rather than concurrently.
2. None of the actions in the Hearts domain is defined recursively, so one instance of an action can't occur as a proper sub-event of another.

These two properties of the domain -- or more precisely, of *the way the domain is modelled* by FOO's knowledge base of concept definitions -- can be clarified by showing what would happen if they were violated:

1. If the domain model allowed two cards to be taken simultaneously, then the condition (DURING [TAKE (TRICK-WINNER) C3] [TAKE ME C1]) would not necessarily imply C3 = C1.

2. Suppose taking a sequence of cards were defined recursively as

```
TAKE-CARDS =
  (LAMBDA (P S)
     (=> (NOT (EMPTY S))
         (SCENARIO
            (TAKE P (FIRST S))
            (TAKE-CARDS P (EXCEPT (FIRST S) S)))))
```

Then (DURING [TAKE-CARDS (TRICK-WINNER) S1] [TAKE-CARDS ME S2]) would not necessarily imply S1 = S2.

In short, the logical status of the partial matching transformation may depend on subtle properties not only of the domain but of the way in which it's modelled in the knowledge base, and may in fact be quite difficult to determine.

Even when partial matching demonstrably does *not* preserve logical equivalence, it can still be useful to use it as though it did. Consider, for example, this reduction from DERIV11:

```
        (=> [VOID P1 (SUIT-LED)] [VOID P0 S0])
  11:11 --- [REDUCE by RULE43] --->
        (AND [= P1 P0] [= (SUIT-LED) S0])
```

One can invent game situations in which P1's being void in the suit led implies that a different player P0 is void in suit S0. Thus the new expression is not a *logically* necessary condition for the original one. However, it is a *heuristically* necessary condition in the sense that it applies to all but a few perverse situations. Thus treating partial matching as an equivalence-preserving transformation may be *logically incorrect* but *empirically valid, i.e.*, lead to good task performance. In this sense partial matching can be used to make *plausible inferences* of the form "if relation R holds between two similar expressions, their corresponding arguments are equal."

A complete operationalization system that used such unproven inferences would need some mechanism to evaluate their empirical validity and identify incorrect inferences with adverse effects. A proposed mechanism for identifying such inferences is *failure-driven learning* (§8.1.7) [Hayes-Roth 81b]. The idea is to remember the inferences made in the course of deriving plans used in performing a task. When a plan fails to achieve its stated goal in a particular task situation, the faulty inference is identified by evaluating each proposition in the plan's derivation *in the context of the specific situation*. (This is much easier than analyzing the general logical validity of each inference -- essentially theorem-proving.) The plan is then corrected to avoid similar failures, *e.g.*, by suitably restricting the conditions in which the plan is used. This approach focuses attention on exactly those incorrect inferences with adverse effects, rather than continously monitoring the empirical validity of every inference made during operationalization.

5.3.4. Partial Matching Rules

RULE43 is only one of several partial matching rules in FOO; it reduces relations between expressions based on n-ary functions. Other rules handle expressions involving quantifiers and sets. The rules are listed below, together with some of the expressions to which they were applied.

RULE30: (R [f S_1] [f S_2]) -> T if (subset S_1 S_2),
where f is monotonic w.r.t. partial-ordering R, *i.e.*, (R [f S] [f S']) if S is a subset of S'

```
13:23 (SUBSET [INTERVALS-OF PROJECT1]
              [INTERVALS-OF (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))])
----> T
since (SUBSET PROJECT1 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))))

13:46 (=< [RANGE PROJECT1]
          [RANGE (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))])
----> T
since (SUBSET PROJECT1 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))))

13:96 (=< [# (SET-OF X3 PROJECT1 (= X3 CLIMAX1))]
          [# (SET-OF X4 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))
                (= X4 CLIMAX1))])
----> T
since (SUBSET [SET-OF X3 PROJECT1 (= X3 CLIMAX1)]
              [SET-OF X4 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))
                 (= X4 CLIMAX1)])
```

RULE43: (R [f e_1 ... e_n] [f e_1' ... e_n']) -> (and [= e_1 e_1'] ... [= e_n e_n']) if R is reflexive

```
3:52 (=> [MOVE C4 (HAND P1) POT] [MOVE C3 (HAND (QSO)) LOC3])
---> (AND [= C4 C3] [= (HAND P1) (HAND (QSO))] [= POT LOC3])

3:55 (= [HAND P1] [HAND (QSO)])
---> (AND [= P1 (QSO)])

5:12 (DURING [TAKE (TRICK-WINNER) C1] [TAKE ME QS])
---> (AND [= (TRICK-WINNER) ME] [= C1 QS])
```

RULE91: (R [f e_1 ... e_n] [f e_1' ... e_n']) -> T if (and [= e_1 e_1'] ... [= e_n e_n']),
where R is reflexive

RULE91 resembles RULE43 but explicitly treats the conjoined equalities as a sufficient condition.

```
2:48 (=> [HAS P1 C2] [HAS P5 C11])
---> T
since (AND [= P1 P5] [= C2 C11])

2:62 (=> [= ((PROJECT CARD-OF (PLAYERS)) ME)
            (HIGHEST-IN-SUIT-LED (PROJECT CARD-OF (PLAYERS)))]
         [= (CARDS-PLAYED1 ME) (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)])
---> T
since (AND [= ((PROJECT CARD-OF (PLAYERS)) ME) (CARDS-PLAYED1 ME)]
           [= (HIGHEST-IN-SUIT-LED (PROJECT CARD-OF (PLAYERS)))
              (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)]
```

```
6:54 (= [SUIT-OF DK] [SUIT-OF (CARD-OF (LEADER))])
---> T
since (AND [= DK (CARD-OF (LEADER))])
```

RULE172: (in (f e) (project f S)) -> (in e S); equivalence-preserving if f is injective

```
4:23 (IN [HAND ME] [PROJECT HAND (PLAYERS)])
---> (IN ME (PLAYERS))

5:23 (IN [CARD-OF ME] [PROJECT CARD-OF (PLAYERS)])
---> (IN ME (PLAYERS))
```

RULE172 is a shortcut for the sequence:

```
(IN (f e) (PROJECT f S))
--- [by RULE14] --->
(EXISTS x S (= [f e] [f x]))
            --- [REDUCE by RULE43, SIMPLIFY by RULE178] --->
(EXISTS x S (= e x))
--- [RECOGNIZE by RULE162] --->
(IN e S)
```

RULE192: (= [Q x S P_x] [Q y S R_y]) -> (= P_y R_y)

```
(= [FORALL P1 (INDICES-OF CARDS-PLAYED1)
      (=> (IN ME (INDICES-OF CARDS-PLAYED1))
          (AND (IN (CARDS-PLAYED1 ME)
                   (CARDS-IN-SUIT-LED CARDS-PLAYED1))
               (=> (= (SUIT-OF (CARDS-PLAYED1 P1)) (SUIT-LED))
                   (NOT (HIGHER (CARDS-PLAYED1 P1)
                                (CARDS-PLAYED1 ME))))))]
   [FORALL P1 (INDICES-OF CARDS-PLAYED1)
      Q1])

(= [EXISTS C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0)]
   [EXISTS X1 (CARDS-IN-HAND P0) (IN X1 S1)])

(= [FORALL I5 (INDICES-OF PROJECT1)
      (NOT (HIGHER (PROJECT1 I5) CLIMAX1))]
   [FORALL I1 (INDICES-OF PROJECT1)
      Q2])
```

RULE322: (=> [f e_1 ... e_n] [exists x S [f e_1' ... e_n']]) -> T if (and [in x S] [= e_1 e_1'] ... [= e_n e_n'])

```
2:30 (=> [HAS P1 C2] [EXISTS P4 (OPPONENTS ME) [HAS P4 C7]])
---> T
since (AND [IN P4 (OPPONENTS ME)] [= P1 P4] [= C2 C7])
```

RULE322 is a shortcut for the sequence:

```
(=> [f e1 ... en] [EXISTS x S [f e1' ... en']])
--- [restructure by RULE220] --->
(EXISTS x S (=> [f e1 ... en] [f e1' ... en']))
            --- [reduce by RULE43] --->
(EXISTS x S (AND [= e1 e1'] ... [= en en']))
--- [eliminate quantifier] --->
(AND [in x S] [= e1 e1'] ... [= en en'])
```

RULE379: (=> [forall x S_1 P_x] [forall y S_2 P_y]) -> T if (subset S_2 S_1)

```
13:22 (=> [FORALL INTERVAL1 (INTERVALS-OF
                            (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))))
             (USABLE-INTERVAL! INTERVAL1)]
          [FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
             (USABLE-INTERVAL! INTERVAL1)])
----> T
since (SUBSET [INTERVALS-OF (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))]
              [INTERVALS-OF PROJECT1])

13:67 (=> [FORALL X1 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))
             (NOT (HIGHER X1 CLIMAX1))]
          [FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1))])
----> T
since (SUBSET [PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))] PROJECT1)
```

RULE380: (= [forall x S_1 Px] [forall y S_2 R_y]) -> (= R_y (or P_y [in y (diff S_2 S_1)]))

```
13:31 (= [FORALL I4 (LB:UB 2 (# PROJECT1))
            (USABLE-INTERVAL!
               (INTERVAL (PROJECT1 (- I4 1)) (PROJECT1 I4)))]
         [FORALL I1 (INDICES-OF PROJECT1) Q1])
----> (= Q1 (OR [USABLE-INTERVAL!
                    (INTERVAL (PROJECT1 (- I1 1)) (PROJECT1 I1))]
                [IN I1 (DIFF (INDICES-OF PROJECT1)
                             (LB:UB 2 (# PROJECT1)))]))
```

RULE388: (subset [set-of x S_1 P_x] [set-of y S_2 P_y]) -> T if (subset S_1 S_2)

```
13:97 (SUBSET [SET-OF X3 PROJECT1 (= X3 CLIMAX1)]
              [SET-OF X4 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))
                 (= X4 CLIMAX1)])
----> T
since (SUBSET PROJECT1 [PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))])
```

5.3.5. Recurrence Equations

Partial matching is also used to *induce the definition of a function from a recurrence equation.* A recurrence equation of the form *h*(e) = (g ... e ...) has a solution *h* = [lambda(x) (g ... x ...)]. Similarly, an equation e = *h*(g ... e ...) has a solution *h* = [lambda(x) (g ... x ...)]$^{-1}$. These solutions are not unique, but are natural in a certain intuitive sense.

Recurrence equations of the form *h*(e) = (g ... e ...) are solved by a rule used to *reformulate an expression as a function of a given quantity* (§4):

RULE315: (reformulate [g ... e ...] e) -> ([lambda (x) (g ... x ...)] e)

The construct (reformulate [g ... e ...] e) can be paraphrased as "solve *h*(e) = (g ... e ...) for *h* and rewrite (g ... e ...) as *h*(e)." RULE315 is used in DERIV2 to reformulate the success criterion for a heuristic search as a predicate on a choice sequence (§3.3.3):

```
(REFORMULATE [DURING (TRICK) (TAKE-POINTS ME)] [CARDS-PLAYED])
```

The condition that e occur in the expression to be reformulated is satisfied by analysis:

```
2:18          --- [by analysis] --->
(REFORMULATE [AND (HAVE-POINTS (CARDS-PLAYED))
                  (= (CARD-OF ME)
                     (HIGHEST-IN-SUIT-LED (CARDS-PLAYED)))]
             [CARDS-PLAYED])
```

Then RULE315 is applied with e = (CARDS-PLAYED), x = CARDS-PLAYED1 to construct the desired function:

```
2:19 --- [by RULE315] --->
([LAMBDA (CARDS-PLAYED1)
    (AND (HAVE-POINTS CARDS-PLAYED1)
         (= (CARD-OF ME) (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)))]
 (CARDS-PLAYED))
```

RULE315 is also used in DERIV13:

```
(REFORMULATE [LEGAL-CANTUS! (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))]
             [PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))])
13:15 --- [by RULE315] --->
([LAMBDA (PROJECT1) (LEGAL-CANTUS! PROJECT1)]
 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))))
```

Recurrence equations of the form *h*(e) = (g ... e ...) are solved by

RULE211: (= (g ... e ...) (h e')) -> (= h (lambda (x) (g ... x ...))) if e = e',
where e and e' are both of the form (f ...)

Recurrence equations of the form e = *h*(g ... e ...) are solved by

RULE212: (= e [h (g ... e' ...)]) -> (= h [inverse (lambda (x) (g ... x ...))]) if e = e',
where e and e' are both of the form (f ...)

Since such recurrence equations can have non-unique solutions, these rules are sound rather than valid. They are sufficiently general to handle equations where the recurring term appears in two different forms whose equality can be verified by analysis. This feature is illustrated in DERIV9 by the solution of the following equation, where H1 is the unknown function:

```
(= [EXISTS C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0)]
   [H1 (DISJOINT (CARDS-IN-HAND P0) S1)])
```

The condition that e and e' share the form (f ...) is satisfied by expanding e' = (DISJOINT ...):

```
9:3-4  --- [ELABORATE by RULE124] --->
(= [EXISTS C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0)]
   [H1 (NOT (EXISTS X1 (CARDS-IN-HAND P0) (IN X1 S1)))])
```

The condition e = e' is verified by partial-matching e and e' and solving for the variable S1:

```
              (= [EXISTS C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0)]
                 [EXISTS X1 (CARDS-IN-HAND P0) (IN X1 S1)])
9:6-12        --- [REDUCE by RULE192, analysis] --->
              T
S1 <- (CARDS-IN-SUIT S0)
```

FOO generates a new variable name EXISTS1 and solves for H1:

```
9:14  --- [REDUCE by RULE215] --->
(= H1 [INVERSE (LAMBDA (EXISTS1) (NOT EXISTS1))])
```

The solution is simplified by rewriting (lambda (x) (f x)) as f:

```
9:15  --- [SIMPLIFY by RULE215] --->
(= H1 [INVERSE NOT])
```

The solution could be further simplified to H1 = NOT based on the knowledge that the function NOT is its own inverse. This was unnecessary for the purposes of the derivation but could be done by adding the relation "INVERSE of NOT is NOT" to FOO's semantic network (§1.3.2.2), and adding a general rule "(inverse f) -> g, where g is the inverse of f."

5.3.6. Partial Matching: Summary

Partial matching can be summarized in terms of *why* to do it, *how* to do it, and *what* it produces:

1. Partial matching is a powerful technique for eliminating troublesome concepts (*e.g.*, the undefined relation DURING) from an expression. Even when the concepts in the expression have definitions (*e.g.*, the action TAKE), partial matching can still be useful: by exploiting concept properties instead of expanding definitions, it facilitates high-level analysis.
2. Partial matching typically transforms a relation between similar (*i.e.*, based on the same function) expressions to a conjunction of equalities. The usual partial matching scenario consists of the following steps:

   a. *Reformulate* an expression in terms of the relation.

   b. *Translate* its arguments in terms of a common function.

   c. *Restructure* the expression into the right form.

   d. *Match* the corresponding arguments.

   e. *Simplify* the result into a useful form.

3. The resulting expression is generally a *sufficient* condition on the original predicate; whether it's a *necessary* condition as well may depend on subtle properties of the task domain and the way it is modelled by the knowledge base. Even when partial matching is not a *logically* valid transformation, treating it as one may be *empirically* justified by the performance of the resulting operationalizations. Moreover, logical *soundness* suffices for certain problems, such as finding a natural solution to a *recurrence equation*.

5.4. Variables, Assumptions, and Finding Examples

In FOO's transformational organization, most of the problem-solving state is encoded in the current expression (or stack of expressions, since subgoaling is permitted). Communication between different branches of the problem-solving process can be a bit awkward within this organization. I therefore added mechanisms to permit global communication via variables and assumptions. For instance, an assumption made in the course of solving a subproblem is stored on a global list of assumptions, from which it can be retrieved and used to simplify other subproblems. Similarly, the binding assigned to a variable in the course of solving a subproblem can be used to simplify other subproblems where the variable occurs. This provides a natural way to express problems that don't readily fit the transformational paradigm, *e.g.*, the problem of assigning values to the arguments of a function in such a way as to make the resulting expression satisfy some property.

A potential refinement would be a mechanism for associating variables and assumptions with different *contexts*, as in partitioned semantic networks [Hendrix 75] and truth maintenance systems [Doyle 81]. Such a mechanism could keep track of alternative assumptions and variable bindings by putting them in different contexts. As implemented, FOO has no mechanism for abandoning one assumption in favor of another, backtracking from a wrong guess about the value of a variable, or investigating multiple possibilities in parallel if they involve conflicting assumptions or variable assignments. This limitation is not serious in FOO's user-guided mode, but would have to be rectified in a more autonomous system (§8.1.1).

This section presents rules that solve for variables, plug in the results, make assumptions, use them, and find elements of intensionally defined sets. (§5.4.1) describes FOO's mechanisms for solving variables and plugging in their values. (§5.4.2) proposes a way to solve inequalities by assuming the compared expressions can be expressed in terms of the same concept. (§5.4.3) presents rules for making and using assumptions. (§5.4.4) discusses how to find an example element of an intensionally defined set. This is closely related to solving for the value of a variable: example-finding can be represented as solving for x in an equation of the form (in x S). The key to example-finding is knowing where to look for plausible candidates; some of FOO's rules try the assumption list -- another connection between example-finding and global communication. Finally, (§5.4.5) classifies FOO's communication rules according to the source and destination of the information they propagate.

5.4.1. Solving for Variables

Solving an equation is the same as proving a proposition except that besides the usual proof operations, the variables in the equations can be assigned values; any consistent assignment that makes the equation a true proposition constitutes a solution. Equations are solved in FOO with the usual apparatus for proving propositions, plus two rules that solve for variables, and a rule that plugs in the value of a solved variable:

RULE194: (= v e) -> T, setting v to e, if v is an unbound variable

RULE271: (= e v) -> T, setting v to e, if v is an unbound variable

RULE127: v -> e if variable v has been set to e; e need not be a constant

The folowing list shows some of the variable assignments made and used by these rules. Notice that a variable may be assigned not only a constant, but an expression, another variable, or an expression containing unbound variables. Thus global variables can be used to propagate *partial information.*

```
6:51 CARD1 <- OK -- constant

3:56 P1 <- (QSO) -- expression

3:53 C4 <- C3 -- variable

2:84 Q1 <- (=> (NOT (AFTER ME P1))
               (AND [IN (CARDS-PLAYED1 ME)
                        (CARDS-IN-SUIT-LED CARDS-PLAYED1)]
                    [=> (= (SUIT-OF (CARDS-PLAYED1 P1)) (SUIT-LED))
                        (NOT (HIGHER (CARDS-PLAYED1 P1)
                                     (CARDS-PLAYED1 ME)))])) -- condition

9:16 H1 <- (INVERSE NOT) -- function-valued expression

9:13 S1 <- (CARDS-IN-SUIT S0) -- expression containing independent variable

11:3 C1 <- (CARD-OF P1) -- expression containing quantifier variable

13:38 Q1 <- (OR [USABLE-INTERVAL! (INTERVAL (PROJECT1 (- I1 1))
                                            (PROJECT1 I1))]
                [= I1 1])) -- quantified condition

13:75 Q2 <- (NOT (HIGHER (PROJECT1 I1) CLIMAX1)) -- musical constraint
```

5.4.2. Solving Inequalities

Solving (= v e) for v is trivial. It's harder to solve for v based on a comparison (R v e) when R is not the equality relation, but one approach for doing so is to

Assume compared expressions can be expressed in terms of the same function.

This idea is expressed in a variant of RULE271 proposed in (§3.5.5.5):

RULE271a: (R [f e_1 ... e_n] e) -> (R [f e_1 ... e_n] [f v_1 ... v_n]), assuming e = (f v_1 ... v_n)

RULE271a would help reduce the goal (=< [MELODIC-RANGE PROJECT1] (MAJOR 10)) of keeping the melodic range of a generated tone sequence PROJECT1 within a major tenth to the problem of choosing and enforcing *a priori* bounds on its lowest and highest notes. The first step in such a reduction would plug in the definition of MELODIC-RANGE:

```
(=< [MELODIC-RANGE PROJECT1] (MAJOR 10))
    --- [ELABORATE by RULE124] --->
(=< [SIZE (INTERVAL (LOWEST PROJECT1) (HIGHEST PROJECT1))]
    (MAJOR 10))
```

Applying RULE271a with R = =<, f = SIZE, e_1 = (INTERVAL ...), e = (MAJOR 10), and v_1 = INTERVAL1 would suggest assuming (MAJOR 10) to be the SIZE of something, since it's compared to (SIZE (INTERVAL ...)) by the =< relation:

```
(=< [SIZE (INTERVAL (LOWEST PROJECT1) (HIGHEST PROJECT1))]
    (MAJOR 10))
--- [assume (MAJOR 10) = (SIZE INTERVAL1) by RULE271a] --->
(=< [SIZE (INTERVAL (LOWEST PROJECT1) (HIGHEST PROJECT1))]
    [SIZE INTERVAL1])
```

The assumption that (MAJOR 10) is the SIZE of an interval allows the problem to be reduced:

```
(=< [SIZE (INTERVAL (LOWEST PROJECT1) (HIGHEST PROJECT1))]
    [SIZE INTERVAL1])
--- [REDUCE by RULE284] --->
(SUBSET [INTERVAL (LOWEST PROJECT1) (HIGHEST PROJECT1)]
        INTERVAL1)
```

Here INTERVAL1 denotes the "something" v_1 whose SIZE is (MAJOR 10). The fact that INTERVAL1 is compared to (INTERVAL (LOWEST PROJECT1) (HIGHEST PROJECT1)) by the SUBSET relation suggests that INTERVAL1 is the interval between two things, call them LOWEST1 and HIGHEST1. This inference is given by RULE271a with R = SUBSET, f = INTERVAL, e_1 = (LOWEST PROJECT1), e_2 = (HIGHEST PROJECT1), e = INTERVAL1, v_1 = LOWEST1, v_2 = HIGHEST1:

```
(SUBSET [INTERVAL (LOWEST PROJECT1) (HIGHEST PROJECT1)]
        INTERVAL1)
--- [assume INTERVAL1 = (INTERVAL LOWEST1 HIGHEST1) by RULE271a] --->
(SUBSET [INTERVAL (LOWEST PROJECT1) (HIGHEST PROJECT1)]
        [INTERVAL LOWEST1 HIGHEST1])
```

Explicitly designating LOWEST1 and HIGHEST1 as notes forming an INTERVAL whose SIZE is (MAJOR 10) makes it possible to satisfy the original constraint on the MELODIC-RANGE of the generated sequence PROJECT1 by making them *a priori* bounds on the lowest and highest notes:

```
(SUBSET [INTERVAL (LOWEST PROJECT1) (HIGHEST PROJECT1)]
        [INTERVAL LOWEST1 HIGHEST1])

--- [by analysis based on knowledge about intervals] --->
(AND [NOT (LOWER (LOWEST PROJECT1) LOWEST1)]
     [NOT (HIGHER (HIGHEST PROJECT1) HIGHEST1)])

--- [by analysis based on transitivity of LOWER, HIGHER] --->
(FORALL NOTE1 PROJECT1
   (AND [NOT (LOWER NOTE1 LOWEST1)]
        [NOT (HIGHER NOTE1 HIGHEST1)]))
```

Thus RULE271a restricts the problem (R [f ...] e) by adding the plausible assumption that the quantity e to which (f ...) is compared can also be expressed in the form (f ...). The revised problem (R [f ...] [f ...]) can then be reduced by various analysis methods based on knowledge about homomorphisms, transitivity, interval arithmetic, etc.

5.4.3. Propagating Assumptions

FOO maintains a global (with respect to a derivation) list of assumptions of the form

Assume that expression e has value v.

The value need not be a constant. To make an assumption, FOO adds e = v to the list. To assume an arbitrary proposition P is true (or false), FOO adds e = T (or e = nil) to the list. The assumption can be applied by substituting v for e later in the problem. The rules used to make and retrieve explicit assumptions appear below, each one followed by examples of its use.

RULE15: (partition S_1 ... S_n) -> (union S_1 ... S_n) assuming (disjoint S_1 ... S_n)

```
(PARTITION (HANDS (PLAYERS)) (PILES (PLAYERS)) (SET DECK POT HOLE))
```

RULE15 unpacks the embedded assertion (disjoint S_1 ... S_n) and makes it an explicit assumption.

RULE157: (=> P Q) -> Q assuming P

```
(=> [IN QS (CARDS-PLAYED)]
    [HIGHER QS (CARD-OF ME)])
```

This assumes that QS will be played, so as to plan accordingly (§2.9.2.2).

RULE221: P > T assuming P, where P is a proposition to be verified

```
(SUBSET PROJECT1 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))))
```

This proposition is an instance of a general fact about heuristic search (§3.1), namely that every partial path is a prefix of the choice sequence. FOO has no mechanism for indexing such facts under the methods they describe. Such an indexing mechanism might help a complete operationalization system retrieve relevant facts quickly instead of examining all rules.

RULE302: (and ... P ... [... P ...] ...) -> (and ... P ... [... T ...] ...)

```
5:42 (IN QS (CARDS-PLAYED)) ---> T

14:56 (HIGHER (HIGHEST (CHANGE CANTUS-1)) (HIGHEST CANTUS-1)) ---> T

14:74 (HIGHER (NEXT NOTE) (HIGHEST CANTUS-1)) ---> T
```

RULE302 uses one branch of a conjunction to simplify another by treating it as a premise. As implemented, RULE302 searches the expression in which P occurs, and any higher-level goals on the stack, to see if P occurs *anywhere* else; the user must decide whether it occurs as a premise.

RULE342: (= c e) -> T assuming e = c, where c is a constant

```
3:26 (= S (SUIT-LED)) ---> T
```

RULE346: e -> v if e = v was assumed earlier

```
3:62 (SUIT-LED) ---> S

3:84 (LEADING (QSO)) ---> NIL

4:17 (DISJOINT (HANDS (PLAYERS)) (PILES (PLAYERS)) (SET DECK POT HOLE))
---> T

6:55 (CARD-OF (LEADER)) ---> DK

8:12 (IN C1 (CARDS)) ---> T

13:47 (SUBSET PROJECT1 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))) ---> T
```

With two exceptions, these assumptions were generated by rules described in this section. To simulate the retrieval of situation-specific information from a state model, I hand-coded the assumption (CARD-OF (LEADER)) = DK for the dynamic operationalization (§8.1.6) variant of DERIV6. To simulate typed variables, I added the assumption (IN C1 (CARDS)) in DERIV8.

RULE349: (=> P Q) -> T assuming P = nil

```
3:80 (=> [LEADING (QSO)]
         [OR (CAN-LEAD-HEARTS (QSO)) (NEQ (SUIT-OF C3) H))])
---> T
```

This rules out the case where player (QSO) leads, so as to pursue the case where (QSO) follows.

RULE351: (achieve P) -> (achieve (and [= e_1 v_1] ... [= e_n v_n])),
where e_1 = v ... e_n = v_n are the assumptions used in reducing the original goal to P

```
3:88 (ACHIEVE (PLAY-SPADE (QSO)))
---> (ACHIEVE (AND [= (LEADING (QSO)) NIL] [= (SUIT-LED) S]))
```

RULE351 is only valid when the assumptions are in fact strong enough to imply P. In the example shown, this implication follows from the rules of the game.

5.4.4. Finding Examples

A problem similar to solving for the value of a variable is that of finding an element of an intensionally described set. This is a difficult problem in general, but FOO has some special-case rules that work in simple cases. These rules are listed below with examples of their use.

RULE84: (... [in c S] ... [find-elt S'] ...) -> (... [in c S]... c ...) if (in c S'), where c is a constant

RULE84 proposes c as a possible element of a set S' if the current expression says that c is in S. The implementation requires that (in c S) occur as a premise. For example, in DERIV5, the condition (IN QS (CARDS-PLAYED)) occurs as such a premise in the goal

```
(=> [IN QS (CARDS-PLAYED)]
    (HIGHER [FIND-ELT (CARDS-IN-SUIT-LED (CARDS-PLAYED))]
            (CARD-OF ME)))
```

This suggests the guess (in c S'), verified by subsequent analysis:

```
             (IN QS (CARDS-IN-SUIT-LED (CARDS-PLAYED)))
5:40-41      --- [by analysis] --->
             (AND [IN QS (CARDS-PLAYED)] [IN-SUIT-LED QS])

                  (IN QS (CARDS-PLAYED))
5:42              --- [by premise] --->
                  T

             (AND T [IN-SUIT-LED QS])
5:43-48      --- [by analysis] --->
             T
```

The constant QS is used as the desired member of S, reducing the goal to (... [in c S] ... c ...):

```
(=> [IN QS (CARDS-PLAYED)]
    (HIGHER QS (CARD-OF ME)))
```

Note the use of the premise in verifying the guess.

RULE109: (not (and ... [forall x S P_x] ...)) -> (=> [and ...] [not $P_{[\text{find-elt } S]}$])

To negate a universally quantified predicate, find a counter-example.

In DERIV5 and DERIV6, RULE109 suggests finding a card played in the suit led that's higher than (CARD-OF ME):

```
(NOT (AND [IN QS (CARDS-PLAYED)]
          [FORALL X1 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
             (NOT (HIGHER X1 (CARD-OF ME)))]))

5:35 --- [by RULE109] --->
(=> [AND (IN QS (CARDS-PLAYED))]
    [NOT (NOT (HIGHER [FIND-ELT (CARDS-IN-SUIT-LED (CARDS-PLAYED))]
                      (CARD-OF ME)))])

5:37-38 --- [SIMPLIFY by RULE178, RULE88] --->
(=> [IN QS (CARDS-PLAYED)]
    [HIGHER [FIND-ELT (CARDS-IN-SUIT-LED (CARDS-PLAYED))]
            (CARD-OF ME)])

(NOT (AND [IN-SUIT-LED (CARD-OF ME)]
          [FORALL X1 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
             (NOT (HIGHER X1 (CARD-OF ME)))]
          [TRICK-HAS-POINTS]))

6:40 --- [by RULE109] --->
(=> [AND (IN-SUIT-LED (CARD-OF ME)) (TRICK-HAS-POINTS)]
    [NOT (NOT (HIGHER [FIND-ELT (CARDS-IN-SUIT-LED (CARDS-PLAYED))]
                      (CARD-OF ME)))])

6:41 --- [SIMPLIFY by RULE88] --->
(=> [AND (IN-SUIT-LED (CARD-OF ME)) (TRICK-HAS-POINTS)]
    [HIGHER [FIND-ELT (CARD-IN-SUIT-LED (CARDS-PLAYED))]
            (CARD-OF ME)])
```

RULE142: (P ... [find-elt S] ...) -> (and [in x S] P_x), where x is a new variable

In DERIV6, RULE142 reformulates a problem of example-finding as solving an equation for CARD1:

```
(HIGHER [FIND-ELT (CARDS-IN-SUIT-LED (CARDS-PLAYED))]
        (CARD-OF ME))
6:44 --- [in DERIV6 situation-specific variant] --->
(AND [IN CARD1 (CARDS-IN-SUIT-LED (CARDS-PLAYED))]
     [HIGHER CARD1 (CARD-OF ME)])
```

```
6:45-46 --- [by analysis] --->
(AND [IN-SUIT-LED CARD1]
     [IN CARD1 (PROJECT CARD-OF (PLAYERS))]
     [HIGHER CARD1 (CARD-OF ME)])

6:47-48 --- [by RULE364, assumption (CARD-OF (LEADER)) = DK] --->
CARD1 <- DK
```

RULE347: (exists x S P_x) -> T if P_c, where c is a constant occurring in P_x

To satisfy an existentially quantified predicate, find a positive example.

In the example below, RULE347 suggests trying c = QS:

```
3:69 (EXISTS C5 (CARDS) (AND [HAS (OWNER-OF QS) C5] [IN-SUIT C5 S]))
---> T
since (AND [HAS (OWNER-OF QS) QS] [IN-SUIT QS S])
```

RULE364: (in y (project f S)) -> T, setting y to v, if (in e S),
where y is an unbound variable and (f e) = v was assumed earlier

RULE364 finds an element of the set (project f S) by solving the equation (in y (project f S)) for y (§5.4.4), as in the dynamic operationalization (§8.1.6) variant of DERIV6:

```
6:47 (IN CARD1 (PROJECT CARD-OF (PLAYERS)))
---> CARD1 <- DK
since (CARD-OF (LEADER)) = DK
and   (IN (LEADER) (PLAYERS))
```

5.4.5. Information Propagation: Summary

The rules described in this section solve for variables, exploit assumptions, and find examples by *propagating information* among assumptions, problems, and variable bindings.

1. Rules for solving variables (§5.4.1) propagate information:
   a. from an equality in a problem to a global variable binding (RULE194, RULE271)
   b. from a global variable binding to a value substitution in a problem (RULE127)
2. A proposed rule for solving inequalities (§5.4.2) propagates information:
   a. from a comparison in a problem to a restriction on the value of a variable (RULE271a)
3. Rules for using assumptions (§5.4.3) propagate information:
   a. from an embedded assertion to a global assumption (RULE15)
   b. from a problem clause to a global assumption (RULE157, RULE221, RULE342, RULE349)

c. from a problem clause to a value substitution in a problem (RULE302)

d. from a global assumption to a value substitution in a problem (RULE346)

e. from a list of global assumptions to the problem of achieving them (RULE351)

4. Rules for finding examples (§5.4.4) propagate information:

a. from a problem clause to the subproblem of finding an example (RULE109, RULE142)

b. from a problem clause to a suggested example (RULE84, RULE347)

c. from a global assumption to a suggested example (RULE364)

These rules go outside of the expression-transformation paradigm to communicate between different branches of the problem-solving process.

5.5. Recursive Definitions

When it's infeasible to provide a single complete definition for a concept, it may still be possible to give a set of special-case rules that provide a partial definition. For example, consider the function CHOICE-SEQ-OF whose value is the sequence of choice events that occur as part of a given event. It is not immediately clear how to encode in FOO's representation a general definition of the form

```
CHOICE-SEQ-OF = (LAMBDA (EVENT) <...>)
```

Such a definition would tell how to extract the choice event sequence of an *arbitrary* event description, but filling in the body would require a way to refer to the sub-events of EVENT and a predicate that tests whether a given event is a choice event. It is easier to formulate special-case rules that extract the choice event sequence from event descriptions of particular forms, *e.g.*, explicit choice events:

RULE309: (choice-seq-of (choose x S E_x)) -> (list (choose x S E_x))

The choice event sequence of a choice event is the list consisting of that event.

Such a special-case rule may be *recursive*, *i.e.*, its right-hand side may refer to the concept in question:

RULE307: (choice-seq-of (each x S E_x)) -> (apply append (each x S (choice-seq-of E_x)))

A scenario's choice event sequence is the concatenation of its components' choice sequences.

This section shows how such rules are used in FOO to (partially) define three general concepts:

1. the functional dependence of one quantity on another (§5.5.1).
2. the choice event sequence of a given event (§5.5.2).
3. the domain of a function (§5.5.3).

(§5.5.4) lists FOO's rules about these concepts, distinguishing *boundary conditions* from *recursive* rules.

5.5.1. Functional Dependence

(§2.8) presented a calculus of four elements {0, ↑, ↓, ?} for describing how one quantity depends on another. The rules below, taken together, provide a partial definition of the DEPENDENCE function. Examples from DERIV9 follow each rule.

RULE203: (dependence [f e_1 ... e_n] v) ->
(D+ ... [D* (dependence (f x_1 ... x_n) x_i) (dependence e_i v)] ...),
where (D+ ...) includes a term [D* ...] for each e_i in which v occurs (chain rule)

```
(DEPENDENCE [PR-DISJOINT-FORMULA (#CARDS-IN-HAND P0)
                                 (#CARDS-OUT-IN-SUIT S0)
                                 (#CARDS-OUT)]
             S0)
9:45 --- [by RULE203] --->
(D+ [D* (DEPENDENCE (#CARDS-OUT-IN-SUIT S0) S0)
        (DEPENDENCE (PR-DISJOINT-FORMULA #H #S #U) #S)])
```

RULE204: (dependence e v) -> nil if the atom v doesn't occur in e

```
(DEPENDENCE (#CHOOSE #U #H) #S)
9:51 --- [EVAL by RULE204] --->
NIL

(DEPENDENCE #U #S)
9:56 --- [EVAL by RULE204] --->
NIL
```

RULE205: (dependence [f e_1 e_2] v) -> (D+ [dependence e_1 v] [D- (dependence e_2 v)]),
where f(x y) is an increasing function of x and a decreasing function of y

```
(DEPENDENCE [/ (#CHOOSE (- #U #S) #H) (#CHOOSE #U #H)] #S)
9:50 --- [by RULE205] --->
(D+ [DEPENDENCE (#CHOOSE (- #U #S) #H) #S]
    [D- (DEPENDENCE (#CHOOSE #U #H) #S)])

(DEPENDENCE (- #U #S) #S)
9:55 --- [by RULE205] --->
(D+ [DEPENDENCE #U #S] [D- (DEPENDENCE #S #S)])
```

RULE207: (dependence [f e_1 ... e_n] v) -> (dependence e_1 v) if v doesn't occur in e_2 ... e_n,
where f is an increasing function of its first argument

```
(DEPENDENCE [#CHOOSE (- #U #S) #H] #S)
9:54 --- [REDUCE by RULE207] --->
(DEPENDENCE (- #U #S) #S)
```

RULE208: (dependence e e) -> increasing

```
(DEPENDENCE #S #S)
9:57 --- [EVAL by RULE208] --->
INCREASING
```

5.5.2. Choice Sequence Extraction

Rules for extracting the choice event sequence of an event are used in DERIV2 and DERIV13.

RULE307: (choice-seq-of [each x S E_x]) -> (apply append [each x S (choice-seq-of E_x)])

```
2:10 (CHOICE-SEQ-OF [EACH P1 (PLAYERS) (PLAY-CARD P1)])
---> (APPLY APPEND [EACH P1 (PLAYERS) (CHOICE-SEQ-OF (PLAY-CARD P1))])

13:4 (CHOICE-SEQ-OF [EACH I1 (LB:UB 1 (CANTUS-LENGTH))
                          (CHOOSE-NOTE I1)])
---> (APPLY APPEND [EACH I1 (LB:UB 1 (CANTUS-LENGTH))
                         (CHOICE-SEQ-OF (CHOOSE-NOTE I1))])
```

RULE308: (choice-seq-of S) -> nil if S contains no choice events

```
2:7 (CHOICE-SEQ-OF (TAKE-TRICK (TRICK-WINNER))) ---> NIL
```

RULE309: (choice-seq-of (choose x S E_x)) -> (list (choose x S E_x))

```
2:12 (CHOICE-SEQ-OF [CHOOSE (CARD-OF P1) (LEGALCARDS P1)
                           (PLAY P1 (CARD-OF P1))])
---> (LIST [CHOOSE (CARD-OF P1) (LEGALCARDS P1) (PLAY P1 (CARD-OF P1))])

13:6 (CHOICE-SEQ-OF [CHOOSE (NOTE I1) (TONES) (NOTE I1)])
---> (LIST [CHOOSE (NOTE I1) (TONES) (NOTE I1)])
```

FOO knows that CHOICE-SEQ-OF is a homomorphism from events to choice sequences with respect to the "addition" operators SCENARIO and APPEND, *i.e.*, that the homomorphism condition $f(x+y) = f(x) +' f(y)$, or in LISP syntax (f (+ x y)) = (+' (f x) (f y)), is satisfied for f = CHOICE-SEQ-OF, + = SCENARIO, +' = APPEND. This fact is encoded as the semantic relations CHOICE-SEQ-OF is-a HOMOMORPHISM and ADD of SCENARIO is APPEND (§1.3.2). It justifies the reduction

```
(CHOICE-SEQ-OF [SCENARIO (EACH P1 (PLAYERS) (PLAY-CARD P1))
                         (TAKE-TRICK (TRICK-WINNER))])
2:6 --- [REDUCE by RULE284] --->
(APPEND [CHOICE-SEQ-OF (EACH P1 (PLAYERS) (PLAY-CARD P1))]
        [CHOICE-SEQ-OF (TAKE-TRICK (TRICK-WINNER))])
```

This reduction is performed by

RULE284: (f ... [g e_1 ... e_n] ...) -> (g' [f ... e_1 ...] ... [f ... e_n ...]),
where the function (lambda (x) (f ... x ...)) is a homomorphism with respect to g and g'

Thus knowledge about a function like CHOICE-SEQ-OF is encoded not only in rules that mention it by name, but also as facts exploited by rules about more general concepts like HOMOMORPHISM. Thus any rule that applies to a function can be considered part of its definition.

5.5.3. Domain of a Function

Rules for determining the domain of a variable in an expression are used in DERIV9. The notation (domain x e) means "the domain of x as it is used in e"; the notation (domain f) means "the domain of the function f." The following rules, shown with examples of their use, simulate a kind of type inheritance.

RULE195: (domain x [f ... e ...]) -> (domain x e) if x occurs in e

```
9:9 (DOMAIN C1 [= (SUIT-OF C1) S0])
--> (DOMAIN C1 (SUIT-OF C1))
```

RULE196: (domain x (f x)) -> (domain f)

```
9:10 (DOMAIN C1 (SUIT-OF C1))
---> (DOMAIN SUIT-OF)
```

RULE197: (domain suit-of) -> (cards) -- *Hearts-specific fact* (§1.3.3)

```
9:11 (DOMAIN SUIT-OF)
---> (CARDS)
```

5.5.4. Recursive Definitions: Summary

In FOO, sets of special-case rules (partially) define concepts that are difficult to encode as lambda expressions. Each rule set includes *boundary condition* rules:

RULE204: (dependence e v) -> nil if the atom v doesn't occur in e

RULE208: (dependence e e) -> increasing

RULE308: (choice-seq-of S) -> nil if S contains no choice events

RULE309: (choice-seq-of (choose x S E_x)) -> (list (choose x S E_x))

RULE197: (domain suit-of) -> (cards)

Each set also contains *recursive* rules whose right-hand side mentions the concept:

RULE203: (dependence [f e_1 ... e_n] v) ->
(D+ ... [D* (dependence (f x_1 ... x_n) x_i) (dependence e_i v)] ...),
where the sum (D+ ...) includes a term [D* (...) (...)] for each e_i in which v occurs (chain rule)

RULE307: (choice-seq-of [each x S E_x]) -> (apply append [each x S (choice-seq-of E_x)])

RULE195: (domain x [f ... e ...]) -> (domain x e) if x occurs in e

Knowledge about a concept can be encoded not only in rules that mention it specifically, but also as facts like "CHOICE-SEQ-OF is a homomorphism" exploited by more general rules like

RULE284: (f ... [g e_1 ... e_n] ...) -> (g' [f ... e_1 ...] ... [f ... e_n ...]) if f is a homomorphism,
where g, g' correspond to +, +' in the homomorphism condition $f(x+y) = f(x) +' f(y)$

5.6. Intersection Search

A powerful technique for solving a problem is to

Reduce the problem to finding a path between two nodes in a precomputable network.

Such a path, if it exists, can be found in bounded time by a general intersection search algorithm. Even if the original problem can't be reduced to an intersection search, the technique is still useful if the existence of such a path is a *necessary condition* for the existence of a solution. In this case, searching for a path and failing to find one is a logically valid (and typically cheap) way to identify insoluble problems. Intersection search may be impractical if the network is expensive to generate, but this problem was not encountered in the examples implemented. The network nodes for these examples correspond to concepts in the knowledge base, and the connections between them are determined by the concept definitions in a simple way that is easy to compute.

This section illustrates the use of intersection search to decide whether an event can occur during a scenario (§5.6.1), to discriminate between deterministic and non-deterministic scenarios (§5.6.2), and to find a common superset of two sets (§5.6.3).

5.6.1. Identifying Impossible Events

A recurring problem in operationalization is to determine whether one kind of event can occur during another, as illustrated by questions that arise in operationalizing Hearts advice:

> Can a player take a card during the part of a trick in which cards are played?
>
> Can a player take points during this part of a trick?
>
> Can a card get into a player's hand while the round is in progress?
>
> Does a player who's void in a suit remain void for the rest of the round?

These questions can be answered quickly by intersection search through a network with nodes representing types of events, and arcs representing "part-of" and "kind-of" relationships, using

> RULE57: (during s e) -> nil if no sub-event of s can be the same kind of event as e

For this method to work, both s and e must be expressed in terms of event types encoded in the network. As implemented, FOO extracts "part-of" and "kind-of" relationships between event types by examining their definitions, rather than encoding them as arcs in a separate data structure, although this would be easy enough to do. Thus the search takes place in a virtual network rather than a physical one. FOO uses recursive procedures to enumerate the sub-events of s and the abstractions of e; if these two sets are disjoint, the rule condition is satisfied.

The *sub-events* of a scenario are defined recursively as follows:

S1. f is a sub-event of (f e_1 ... e_n) unless f is a low-level event like MOVE or BECOME.

S2. If f is defined as (lambda (...) (g ...)), the sub-events of f include those of (g ...).

S3. The sub-events of (scenario e_1 ... e_n) consist of those of each of e_1 ... e_n.

S4. The sub-events of (Q x S E_x) are those of E_x if Q is FOR-SOME, EACH, or CHOOSE.

S5. The sub-events of (while P e) consist of those of e.

The *abstractions* of an event are defined recursively as follows:

A1. f is an abstraction of (f e_1 ... e_n) unless f is SCENARIO or a low-level event.

A2. If f is defined as (lambda (...) (g ...)), the abstractions of f include those of (g ...).

A3. The abstractions of (and e_1 ... e_n) consist of those of each of e_1 ... e_n.

A4. The abstractions of (while P e) consist of those of e.

A5. The abstractions of (Q x S E_x) consist of those of E_x unless Q is EACH.

The rest of this section shows in detail how RULE57 is used to derive four facts:

1. The Queen of spades can't be taken while cards are being played. (§5.6.1.1)
2. Points can't be taken while cards are being played. (§5.6.1.2)
3. Cards can't become "out" while the round is in progress. (§5.6.1.3)
4. Voids can't be undone while the round is in progress. (§5.6.1.4)

The variations exhibited in the examples demonstrate RULE57's generality.

5.6.1.1 Queen Can't be Taken While Cards are Played

RULE57 can be illustrated by showing how it is used in DERIV5 to make the reduction

```
(DURING [EACH P1 (PLAYERS) (PLAY-CARD P1)] [TAKE ME QS]) -> NIL
```

First FOO finds the sub-events of (EACH P1 (PLAYERS) (PLAY-CARD P1)). By S4, these include

```
PLAY-CARD =
   (LAMBDA (P)
      (CHOOSE (CARD-OF P) (LEGALCARDS P)
         (PLAY P (CARD-OF P)))
```

By S2, S4, and S1, the sub-events of PLAY-CARD include

```
PLAY =
   (LAMBDA (P C)
      (MOVE C (HAND P) POT))
```

The event PLAY has no-subevents by the definition above. If MOVE were considered a sub-event, it would be a sub-event of every action in Hearts. Any two actions would have MOVE as a common sub-event, so RULE57 would never apply. MOVE is "low-level" rather than "primitive" since it actually has a definition, namely

```
MOVE =
   (LAMBDA (OBJ ORIG DEST)
      (AND [= (LOC OBJ) ORIG] [BECOME (LOC OBJ) DEST]))
```

This means "OBJ moves from ORIG to DEST when its location changes from ORIG to DEST." The functions LOC and BECOME are *fluents* (implicit functions of time). It is possible to define

```
BECOME =
   (LAMBDA (F C)
      (AND [≠ F C] [= (NEXT F) C]))
```

This means "F becomes C *(at time t)* if F is not C *(at time t)* but the next F is C." (LOC X) means "the location of X *(at time t)*." If time is discrete, it is possible to define the fluent

```
NEXT =
   (LAMBDA (F)
      [LAMBDA (T) (F (+ T 1))])
```

This means "the value of (NEXT F) *(at time t)* is the value of F at time t+1."

In short, MOVE can be elaborated down to a very primitive level. However, the knowledge base of Hearts concepts is deliberately constructed so that all higher-level actions that share a common sub-event have at least one common sub-event at a higher level than MOVE. This means that an intersection search for a common sub-event of two higher-level actions can safely stop at MOVE.

So the only sub-events of (EACH P1 (PLAYERS) (PLAY-CARD P1)) are {PLAY-CARD, PLAY}.

FOO next finds the abstractions of (TAKE ME QS). By A1, these include

```
TAKE = (LAMBDA (P C) (MOVE C POT (PILE C)))
```

The event MOVE doesn't count, since it's a low-level event, so the only abstraction of (TAKE ME QS) is TAKE. Since {PLAY-CARD, PLAY} and {TAKE} are disjoint, the condition of RULE57 is satisfied. FOO deduces that one can't take the Queen during the part of the trick when cards are played.

5.6.1.2 Can't Take Points While Cards are Played

A similar example occurs in DERIV6, where RULE57 is used to make the reduction

```
(DURING [EACH P1 (PLAYERS) (PLAY-CARD P1)] [TAKE-POINTS ME]) -> NIL
```

The abstractions of (TAKE-POINTS ME) are {TAKE-POINTS, TAKE} by A1, A2, A5, and the definition

```
TAKE-POINTS = (LAMBDA (P) (FOR-SOME C (POINT-CARDS) (TAKE P C)))
```

Since these are disjoint from {PLAY-CARD, PLAY}, FOO deduces that one can't take points during the part of the trick in which cards are played.

5.6.1.3 Cards Can't Become Out During Round

DERIV10 includes a proof that a card can't become "out", *i.e.*, get into an opponent's hand, while the round is in progress. "Becoming out" is first reformulated in terms of known actions:

```
(DURING [ROUND-IN-PROGRESS] [CAUSE (OUT X1)])
10:5-9                      --- [by analysis] --->
(DURING [ROUND-IN-PROGRESS] [EXISTS P1 (OPPONENTS ME)
                               (GET-CARD P1 X1 LOC1)])
```

To compute the sub-events of (ROUND-IN-PROGRESS), FOO examines the definitions

```
ROUND-IN-PROGRESS =
  (LAMBDA () (EACH I (TRICK-INDICES) (TRICK)))

TRICK =
  (LAMBDA ()
     (SCENARIO (EACH P (PLAYERS) (PLAY-CARD P))
               (TAKE-TRICK (TRICK-WINNER))))

PLAY-CARD =
  (LAMBDA (P)
     (CHOOSE (CARD-OF P) (LEGALCARDS P)
        (PLAY P (CARD-OF P))))

PLAY =
  (LAMBDA (P C)
     (MOVE C (HAND P) POT))

TAKE-TRICK =
  (LAMBDA (P)
     (EACH C (CARDS-PLAYED) (TAKE P C)))

TAKE =
  (LAMBDA (P C)
     (MOVE C POT (PILE C)))
```

The sub-events of (ROUND-IN-PROGRESS) are {ROUND-IN-PROGRESS, TRICK, PLAY-CARD, PLAY, TAKE-TRICK, TAKE}, but (EXISTS P1 (OPPONENTS ME) (GET-CARD P1 X1 LOC1)) has as its only abstraction

```
GET-CARD = (LAMBDA (P C ORIG) (MOVE C ORIG (HAND P)))
```

Therefore FOO concludes that a card can't become out while a round is in progress.

5.6.1.4 Voids Can't be Undone During the Round

A similar problem occurs in DERIV12: show that a void can't be undone while a round is in progress. Since undoing a void is not a named action, it is reformulated as one:

```
(DURING [ROUND-IN-PROGRESS] [UNDO (VOID P0 S0)])
12:2-10                     --- [by analysis] --->
(DURING [ROUND-IN-PROGRESS] [EXISTS C1 (CARDS)
                                 (AND [GET-CARD P0 C1 LOC1]
                                      [IN-SUIT C1 S0])])
```

The sub-events of (ROUND-IN-PROGRESS) are the same as before, while according to A3 the abstractions of (EXISTS C1 (CARDS) (AND [GET-CARD P0 C1 LOC1] [IN-SUIT C1 S0])) are {GET-CARD, IN-SUIT, =}, based on the above definition of GET-CARD and the definition

```
IN-SUIT = (LAMBDA (C SUIT) (= (SUIT-OF C) SUIT))
```

The predicates IN-SUIT and = are erroneously identified as actions by A3 as it applies to the conjunction (AND [GET-CARD P0 C1 LOC1] [IN-SUIT C1 S0]). Such conjunctions are meaningful since actions are predicates in FOO's semantics (§1.3.1.3), but the routine for finding abstractions doesn't check which conjuncts are actions. This could easily be fixed by changing A3 to

A3'. The abstractions of (and $e_1 \ldots e_n$) consist of those of the actions among $e_1 \ldots e_n$.

At any rate, it causes no trouble in this example: the sets are still disjoint, and FOO concludes that a player who is void in a suit remains void for the rest of the round.

5.6.2. *Identifying Choice-Free Events*

Intersection search helps find the choice events in a scenario by quickly eliminating parts of the scenario in which no choice occurs. This is illustrated in DERIV2, where FOO finds the sequence of choice events in a trick (§3.3.2):

```
(CHOICE-SEQ-OF (TAKE-TRICK (TRICK-WINNER)))
2:7 --- [by RULE308] --->
NIL
```

This transformation is performed by

RULE308: (choice-seq-of s) -> nil if no sub-events of s are choice events

The sub-events of (TAKE-TRICK (TRICK-WINNER)) are {TAKE-TRICK, TAKE}, since

```
TAKE-TRICK =
  (LAMBDA (P)
     (EACH C (CARDS-PLAYED) (TAKE P C)))

TAKE =
  (LAMBDA (P C) (MOVE C POT (PILE P)))
```

A *choice event* is an event of the form (choose x S E_x). Since neither TAKE-TRICK nor TAKE is a choice event, the choice sequence of (TAKE-TRICK (TRICK-WINNER)) is NIL.

5.6.3. Finding a Common Superset

The problem of finding a comon superset of two sets arises in DERIV9 as a sub-problem of applying the disjoint subsets method (§2.3). The two sets are (CARDS-IN-HAND P0) and (CARDS-IN-SUIT S0). The disjoint subsets method requires a common superset *whose size will be known at runtime.* (Problems associated with representing and testing such conditions are discussed in (§8.1.1).) Obviously any two sets have an infinite number of common supersets, of which the smallest is their union. The known-size requirement precludes using the union in this case. Thus the problem is to substitute a set of known size for

```
(COMMON-SUPERSET [CARDS-IN-HAND P0] [CARDS-IN-SUIT S0])
```

The construct (common-superset S_1 S_2) denotes the (non-unique) set-valued solution S of the equation (and [subset S_1 S] [subset S_2 S]). The problem is solved in DERIV9 as follows:

```
(COMMON-SUPERSET [CARDS-IN-HAND P0] [CARDS-IN-SUIT S0])
9:23-24          --- [ELABORATE by RULE124] --->
(COMMON-SUPERSET [SET-OF C2 (CARDS) (HAS P0 C2)]
                 [SET-OF C3 (CARDS) (IN-SUIT C3 S0)])
9:25 --- [by RULE199] --->
(CARDS)
```

Here the effect of an intersection search is *simulated* by

RULE199: (common-superset [set-of x S P_x] [set-of y S R_y]) -> S

It would be straightforward to implement an intersection search procedure for finding common supersets. The associated network would contain a node for each defined set concept. The node for a set defined as (set-of x S P_x) or (filter S P) would be connected by a "superset-of" arc to the node for S. Starting from the nodes for the two given sets S_1 and S_2, the search procedure would follow "superset-of" arcs until the two branches of the search encountered a common node; the corresponding set would be a common superset of S_1 and S_2. Alternatively, the procedure could exhaustively find all defined supersets of each set, and return the intersection of these two collections. Some other process could then choose from the collection of common supersets the one best satisfying problem-specific criteria, such as known size.

5.6.4. Intersection Search: Summary

The key points about intersection search emphasized in this section are:

1. Reducing a problem to intersection search through a precomputed network is a powerful technique, since such a search can be performed efficiently (in bounded time).
2. An unsuccessful intersection search can be very useful as a quick way to rule out a possibility for which the existence of a path is a necessary condition, even when the original problem can't be reduced to an intersection search.
3. The network representation used by an intersection search procedure can be *automatically* extracted from the definitions of the concepts corresponding to the nodes.

FOO's intersection search procedures are not especially efficient or general. To test for the existence of a path between two nodes, FOO exhaustively enumerates the set of nodes accessible to each one and only then checks to see if the two sets overlap. For a large network, it would be more efficient to check for intersections while expanding the accessible-node sets. Moreover, the procedures are specific to particular kinds of problems -- showing an event is impossible and deciding if a scenario involves choice.

In short, the interest here is not in improved search algorithms, but in how methods like intersection search can be mechanically applied to a given problem. I showed how the information required to precompute the nodes and arcs of the network for an intersection search can be mechanically extracted from a set of concept definitions. To make this point, it was sufficient to create virtual networks defined by procedures for finding the neighbors of a node; it was not necessary to physically encode such networks as separate data structures, so I didn't. However, this additional step would be required to exploit the full efficiency of intersection search.

5.7. Problem Separation

A very general strategy for reducing a problem is to

Split the problem into two or more separately solvable subproblems.

Depending on how the subproblems relate to the original problem, it may not even be necessary to solve them all. For example, a proposition can be disproved by splitting it into a conjunction and disproving one conjunct. Similarly, a goal can be achieved by splitting it into a disjunction and achieving one disjunct.

Some problems are solvable by a *divide-and-conquer* approach: solve the subproblems and combine their solutions into a solution of the original problem. Here, solving the original problem requires solving all the subproblems; there may be no alternative if the original problem is not directly solvable by available methods. Examples of divide-and-conquer include splitting a goal into conjunctive subgoals, or breaking it down into alternative contingencies, and generating a separate plan for each. A function might be computed by reformulating it as a sum, evaluating each term, and adding the results. As this example suggests, problem splitting can be used not only to transform a proposition into a conjunction or disjunction, but also to split a non-boolean problem into subproblems related to the original problem by addition or some other non-logical function. However, most of the examples in this section involve logically related subproblems.

Separating a problem into subproblems is a little like chipping away built-up ice in a refrigerator: you find a crack you can fit a chisel into, and then hammer away to widen the crack until the whole thing fractures into pieces. In both cases, the choice of a good starting point depends on the specific problem and affects the ease or difficulty of propagating the split all the way through. In problem separation, the initial split is typically provided by a domain-specific rule or definition, while more general methods are used to propagate it. Actually, defrosting is a much safer way to get rid of built-up ice in your freezer, since the chisel method runs the risk of puncturing a freezer line, which costs more to repair than replacing the refrigerator, as I found out the hard way. I mention this because

1. A dissertation should communicate useful information, and
2. It illustrates a *disadvantage* of empirical discovery relative to analytic methods (§8.1.7).

In general, problem separation works by *splitting* some term in the problem and then *propagating* the split up to successively higher levels of the problem description so as to transform it into a collection of separate subproblems -- *separate* in the sense that different methods can be used on each one, although assumptions made in solving one may constrain the solution of the others (§5.4.3). This *problem splitting scenario* can be described more precisely as follows:

1. *Reformulate* a problem (P ...) in terms of a function f: (P ...) -> (P ... (f ... e ...) ...).
2. *Split* an argument of f into a list of components: (f ... e ...) -> (f ... [g e_1 ... e_n] ...).
3. *Propagate* the split upward: (f ... [g e_1 ... e_n] ...) -> (g' [f ... e_1 ...] ... [f ... e_n ...]).
4. *Analyze* each component (f ... e_i ...) separately.

The general concept of problem splitting has some interesting specializations:

1. Conjunctive subgoaling

a. Partial matching

2. Case analysis

a. Condition factoring

These can be characterized as special cases of the general problem splitting transformation

(f ... e ...) -> (g' [f' ... e_1 ...] ... [f' ... e_n ...])

Conjunctive subgoaling corresponds to the case where g' = AND, and f, f' are predicates P, P':

(P ... [g e_1 ... e_n] ...) -> (and [P' ... e_1 ...] ... [P' ... e_n ...])

If the initial proposition represents a goal to be achieved, each conjunct represents a subgoal. One form of conjunctive subgoaling already discussed is *partial matching* (§5.3), where f is a reflexive binary relation R, and f' is the equality relation:

(R [g e_1 ... e_n] [g e_1' ... e_n']) -> (and [= e_1 e_1'] ... [= e_n e_n'])

Case analysis is problem splitting with g' = OR:

(P ... [g e_1 ... e_n] ...) -> (or [P' ... e_1 ...] ... [P' ... e_n ...])

Each disjunct represents a subclass of the situations characterized by the initial proposition. One kind of case analysis, which I'll call *condition factoring* for want of a better term, involves splitting a problem e into two cases: solving the problem when some condition C is true, and solving it when C is false. If e is a proposition to be evaluated or achieved, factoring it with respect to C corresponds to one of the following transformations:

e -> (and [=> C e] [=> (not C) e])

e -> (or [and C e] [and (not C) e])

e -> (if C e_C $e_{\sim C}$)

Each transformation splits the problem into two separate subproblems. The last one is used if e is not a proposition, since IF, unlike AND and OR, can take expressions other than propositions. The construct (if C e_C $e_{\sim C}$) factors e into two contexts, one assuming C, the other assuming ~C. Unlike AND and OR, IF takes a fixed number of arguments, and their order is important. An alternative syntax without this restriction is the LISP construct (cond [C e_C] [~C $e_{\sim C}$]) [McCarthy 63]. The IF construct corresponds to a special case of COND with two complementary conditions.

The rest of this section is organized as follows. (§5.7.1) illustrates the problem splitting scenario using two detailed examples of case analysis. The next two sections describe condition factoring (§5.7.2) and conjunctive subgoaling (§5.7.3). (§5.7.4) presents FOO's rules for introducing initial problem splits, and (§5.7.5) lists the rules used to propagate such splits. (§5.7.6) briefly discusses the technique of *merging* related subproblems into a single problem. Finally, (§5.7.7) draws some conclusions about problem splitting in general.

5.7.1. Case Analysis

The general scenario for *case analysis* consists of the following steps:

1. *Reformulate* an expression in terms of a predicate R.
2. *Split* an argument of R into a list of components.
3. *Propagate* the split upward to produce a disjunction.
4. *Analyze* each case of the disjunction separately, perhaps *eliminating* it.

This scenario is illustrated in DERIV6 in the analysis of how to avoid taking points during a trick. The problem is split into two cases:

1. Taking points while cards are being played
2. Taking points when the winner takes the trick

The first case is found to be impossible, and the second case is reduced to a simpler condition.

The initial problem is first reformulated in terms of the predicate DURING:

```
(AVOID (TAKE-POINTS ME) (TRICK))
6:2 --- [ELABORATE by RULE124] --->
(ACHIEVE (NOT (DURING [TRICK] [TAKE-POINTS ME])))
```

The first argument to DURING is then *split* into a scenario of two components:

```
                         (TRICK)
6:3                      --- [ELABORATE by RULE124] --->
                         (SCENARIO [EACH P1 (PLAYERS) (PLAY-CARD P1)]
                                   [TAKE-TRICK (TRICK-WINNER)])
```

The predicate on the scenario is reformulated as a disjunction of predicates on the components:

```
                    (DURING [SCENARIO (EACH P1 (PLAYERS) (PLAY-CARD P1))
                                      (TAKE-TRICK (TRICK-WINNER))]
                         (TAKE-POINTS ME))
6:4                 --- [by RULE284] --->
                    (OR [DURING (EACH P1 (PLAYERS) (PLAY-CARD P1))
                                (TAKE-POINTS ME)]
                        [DURING (TAKE-TRICK (TRICK-WINNER))
                                (TAKE-POINTS ME)])
```

This step is performed by a general rule about homomorphisms:

RULE284: (f ... [g e_1 ... e_n] ...) -> (g' [f ... e_1 ...] ... [f ... e_n ...]) if f is homomorphic w.r.t. g, g'

Here f = DURING, g = SCENARIO, e_1 = (EACH P1 (PLAYERS) (PLAY-CARD P1)), e_2 = (TAKE-TRICK (TRICK-WINNER)), and g' = OR.

This transformation *propagates the split upward* one level in the current expression, producing a disjunction whose terms can be analyzed *individually*. One case is *eliminated* by intersection search (§5.6):

```
                        (DURING [EACH P1 (PLAYERS) (PLAY-CARD P1)]
                                [TAKE-POINTS ME])
6:5                     --- [COMPUTE by RULE57] --->
                        NIL
```

The other case is reduced by partial matching (§5.3):

```
                        (DURING [TAKE-TRICK (TRICK-WINNER)]
                                [TAKE-POINTS ME])
6:7-22                  --- [REDUCE by partial matching, analysis] --->
                        (AND [= (TRICK-WINNER) ME] [TRICK-HAS-POINTS])
```

Thus case analysis transforms the initial problem into a disjunction of separately analyzable propositions, each of which is treated using the methods suited to it (intersection search and partial matching, respectively).

Case analysis is also used in DERIV4 in the course of applying the pigeon-hole principle to the problem of deciding whether the Queen is out. The pigeon-hole principle implies that the Queen is in the hand of one of player ME's opponents iff it's not at any other location:

```
(OUT QS)
4:2-6 --- [by analysis] --->
(IN (LOC QS) (PROJECT HAND (OPPONENTS ME)))
4:7 --- [by RULE169] --->
(NOT (IN (LOC QS)
         (DIFF (RANGE LOC) (PROJECT HAND (OPPONENTS ME)))))
```

The initial split in the problem comes from partitioning the set of possible locations:

```
                    (RANGE LOC)
    4:9             --- [ENUMERATE by RULE163] --->
                    (PARTITION (HANDS (PLAYERS))
                               (PILES (PLAYERS))
                               (SET DECK POT HOLE))
```

This step uses a Hearts-specific "fact-rule" (§1.3.3) to simulate a type mechanism and subset hierarchy:

RULE163: (range loc) -> (partition (hands (players)) (piles (players)) (set deck pot hole))

To propagate the split upward, the `DIFF` operator must be eliminated. To do this, a set identity is applied after putting the problem in the right form:

```
              (DIFF (PARTITION (HANDS (PLAYERS))
                               (PILES (PLAYERS))
                               (SET DECK POT HOLE))
                    [PROJECT HAND (OPPONENTS ME)])
    4:10-14   --- [by analysis] --->
              (DIFF (UNION (HANDS (PLAYERS))
                           (PILES (PLAYERS))
                           (SET DECK POT HOLE))
                    [DIFF (HANDS (PLAYERS)) (SET (HAND ME))])
    4:15      --- [by RULE164] --->
              (UNION [DIFF (UNION (HANDS (PLAYERS))
                                  (PILES (PLAYERS))
                                  (SET DECK POT HOLE))
                           (HANDS (PLAYERS))]
                     [INTERSECT (UNION (HANDS (PLAYERS))
                                       (PILES (PLAYERS))
                                       (SET DECK POT HOLE))
                                (SET (HAND ME))])
```

The set identity is expressed by

RULE164: (diff S_1 [diff S_2 S_3]) -> (union [diff S_1 S_2] [intersect S_1 S_3])

RULE164 introduces a two-way split in that the two terms in `(UNION [...] [...])` can be treated separately. The first term is simplified:

```
                    (DIFF (UNION (HANDS (PLAYERS))
                                 (PILES (PLAYERS))
                                 (SET DECK POT HOLE))
                          (HANDS (PLAYERS)))
    4:16-18         --- [SIMPLIFY by RULE19] --->
                    (UNION (PILES (PLAYERS)) (SET DECK POT HOLE))
```

The second term is *factored* into two cases by

RULE165: (intersect S [set c]) -> (if [in c S] [set c] nil)

The two cases correspond to whether or not (HAND ME) is among the enumerated locations:

```
                      (INTERSECT (UNION (HANDS (PLAYERS))
                                        (PILES (PLAYERS))
                                        (SET DECK POT HOLE))
                                 [SET (HAND ME)])
4:20                  --- [by RULE165] --->
                      (IF [IN (HAND ME)
                              (UNION (HANDS (PLAYERS))
                                     (PILES (PLAYERS))
                                     (SET DECK POT HOLE))]
                          [SET (HAND ME)]
                          NIL)
```

The original 3-way split is now propagated into the factoring condition (IN ...):

```
                          (IN (HAND ME)
                              (UNION (HANDS (PLAYERS))
                                     (PILES (PLAYERS))
                                     (SET DECK POT HOLE)))
4:21                      --- [by RULE284] --->
                          (OR [IN (HAND ME) (HANDS (PLAYERS))]
                              [IN (HAND ME) (PILES (PLAYERS))]
                              [IN (HAND ME) (SET DECK POT HOLE)])
```

The first term of this disjunction is true, so the whole disjunction is true. Since the factoring condition is true, the conditional expression reduces to the "true" branch:

```
                      (IF T (SET (HAND ME)) NIL)
4:26                  --- [SIMPLIFY by RULE185] --->
                      (SET (HAND ME))
```

Now that DIFF is gone, a bit of simplification allows the original split to be propagated:

```
          (IN (LOC QS)
              (UNION [UNION (PILES (PLAYERS)) (SET DECK POT HOLE)]
                     [SET (HAND ME)])

4:28-29    --- [SIMPLIFY by RULE177, RULE187] --->
          (IN (LOC QS)
              (UNION (SET DECK POT HOLE (HAND ME)) (PILES (PLAYERS))))

4:31 --- [by RULE284] --->
          (OR [IN (LOC QS) (SET DECK POT HOLE (HAND ME))]
              [IN (LOC QS) (PILES (PLAYERS))])

4:32-33   --- [by RULE182, simplification] --->
          (OR [= (LOC QS) DECK]
              [= (LOC QS) POT]
              [= (LOC QS) HOLE]
              [= (LOC QS) (HAND ME)]
              [IN (LOC QS) (PILES (PLAYERS))])
```

The propagation at step 4:31 is performed by

RULE284: (f ... [g e_1 ... e_n] ...) -> (g' [f ... e_1 ...] ... [f ... e_n ...]) if f is homomorphic w.r.t. g, g'

Here f = IN, g = UNION, g' = OR. The subsequent propagation at step 4:32 is performed by

RULE182: (in e (set e_1 ... e_n)) -> (or [= e e_1] ... [= e e_n])

The problem has now been split into five cases corresponding to the terms of the disjunction. The rest of the derivation treats each one separately, using different methods. The DECK case is eliminated; the POT and (HAND ME) cases are reformulated in terms of observable predicates; the (PILES (PLAYERS)) case is reformulated in terms of a remembered event; and the HOLE case cannot usually be detected, but is unlikely enough to be ignored (assumed false) (§2.9.3):

```
(NOT (OR [IN-POT QS] [AT QS HOLE] [HAS-ME QS] [TAKEN QS]))
```

Problem separation was used more than once in this example. First the problem was split by partitioning (RANGE LOC). Applying the difference-of-differences identity made two copies of this split. The first copy was simplified by removing (HANDS (PLAYERS)) from the partition. The second copy was incorporated in a factoring condition and propagated up to split that condition into a disjunction. A term of the disjunction was proved, reducing the factoring condition to true, and the false branch was discarded. This enabled the original split to be propagated upward through the overall expression so as to form a disjunction of separately treatable cases.

5.7.2. Condition Factoring

Condition factoring is case analysis with two cases, one when some *factoring condition* C is true, the other when C is false, as illustrated in DERIV14:

```
(= (#OCCURRENCES (NEXT (CLIMAX CANTUS-1)) (NEXT CANTUS-1)) 1)
14:8 --- [by RULE276] --->
(OR [AND [NOT (CHANGE (CLIMAX CANTUS-1))]
         [= (#OCCURRENCES (CLIMAX CANTUS-1) (NEXT CANTUS-1)) 1]]
    [AND [CHANGE (CLIMAX CANTUS-1)]
         [= (#OCCURRENCES (NEXT (CLIMAX CANTUS-1))
                          (NEXT CANTUS-1))
          1]])
```

Here C = (CHANGE (CLIMAX CANTUS-1)) is given by (change e) in

RULE276: $P_{(\text{next } e)}$ -> (or [and [not (change e)] P_e] [and [change e] $P_{(\text{next } e)}$])

Factor a goal that depends on an expression's next value according to whether it changes.

In this example, the goal of having the climax tone occur exactly once is factored into two cases (1 and 2) according to whether the climax changes on the next note. Case 1 is itself factored:

```
               (= [#OCCURRENCES (CLIMAX CANTUS-1) (NEXT CANTUS-1)] 1)
14:10-19          --- [by analysis] --->
               (= [+ (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1)
                     (# [SET-OF X1 (LIST (NEXT NOTE))
                           (= X1 (CLIMAX CANTUS-1))])]
                  1)

14:20-21                 --- [by RULE121, SIMPLIFY by RULE178] --->
               (= [+ (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1)
                     (# [IF (= (NEXT NOTE) (CLIMAX CANTUS-1))
                            (SET (NEXT NOTE))
                            NIL])]
                  1)
```

The factoring condition `(= (NEXT NOTE) (CLIMAX CANTUS-1))` is given by P_{e_1} in

RULE121: (set-of x (collection e_1 ... e_n) P_x) -> (union [if P_{e_1} {e_1} nil] ... [if P_{e_n} {e_n} nil])

Propagating the conditional split up one level permits some simplification:

```
                     (# [IF (= (NEXT NOTE) (CLIMAX CANTUS-1))
                            (SET (NEXT NOTE))
                            NIL])
14:22                --- [by RULE287] --->
                     (IF (= (NEXT NOTE) (CLIMAX CANTUS-1))
                         [# (SET (NEXT NOTE))]
                         [# NIL])
14:23-24                 --- [SIMPLIFY by RULE288, RULE289] --->
                     (IF (= (NEXT NOTE) (CLIMAX CANTUS-1))
                         1
                         0)
```

The split is propagated by

RULE287: (f ... [if C e_C $e_{\sim C}$] ...) -> (if C [f ... e_C ...] [f ... $e_{\sim C}$ ...])

This rule is applied again to propagate the split up another level:

```
               (= (+ (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1)
                     [IF (= (NEXT NOTE) (CLIMAX CANTUS-1)) 1 0])
                  1)
14:25          --- [by RULE287] --->
               (IF (= (NEXT NOTE) (CLIMAX CANTUS-1))
                   [= (+ (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1) 1]
                   [= (+ (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 0) 1])
```

This adds enough context to simplify one branch:

```
                (= (+ (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 0) 1)
14:26              --- [SIMPLIFY by RULE343] --->
                (= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1)
```

The added context also makes it possible to eliminate the other branch:

```
                (= (+ (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1) 1)
14:27           --- [REDUCE by RULE290] --->
                (= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 0)
14:29-30        --- [by analysis] --->
                (NOT (IN (CLIMAX CANTUS-1) CANTUS-1))
14:31-34        --- [by analysis] --->
                NIL
```

The surviving branch is conjoined with the negation of the factoring condition:

```
            (IF (= (NEXT NOTE) (CLIMAX CANTUS-1))
                NIL
                (= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1))
14:36       --- [RECOGNIZE by RULE294] --->
            (AND [= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1]
                 [NOT (= (NEXT NOTE) (CLIMAX CANTUS-1))])
```

This reduction is performed by

RULE294: (if C nil P) -> (and P [not C])

The idea is that the factoring condition C must be false to satisfy the condition (if C nil P).

Condition factoring is also used to analyze Case 2, where the climax changes when the next note is added:

```
        (AND [CHANGE (CLIMAX CANTUS-1)]
             [= (#OCCURRENCES (NEXT (CLIMAX CANTUS-1))
                              (NEXT CANTUS-1))
              1])
14:53-63 --- [by analysis] --->
        (AND [HIGHER (NEXT NOTE) (CLIMAX CANTUS-1)]
             [= (+ (#OCCURRENCES (NEXT NOTE) CANTUS-1)
                   (# [SET-OF X4 (LIST (NEXT NOTE))
                         (= X4 (NEXT NOTE))]))
                1])

                   (# [SET-OF X4 (LIST (NEXT NOTE))
                         (= X4 (NEXT NOTE))]))
14:64-65              --- [by RULE121, SIMPLIFY by RULE178] --->
                   (# [IF (= (NEXT NOTE) (NEXT NOTE))
                          (SET (NEXT NOTE))
                          NIL])
```

The suggested factoring condition is P_{e_1} in

RULE121: (set-of x (collection $e_1 \ldots e_n$) P_x) -> (union [if P_{e_1} {e_1} nil] ... [if P_{e_n} {e_n} nil])

This condition reduces immediately to true:

```
                                (= (NEXT NOTE) (NEXT NOTE))
    14:66                       --- [EVAL by RULE179] --->
                                T
```

This makes it possible to eliminate the second branch without propagating the split:

```
                            (IF T (SET (NEXT NOTE)) NIL)
    14:67                   --- [SIMPLIFY by RULE185] --->
                            (SET (NEXT NOTE))
```

This in turn enables considerable simplification:

```
        (AND [HIGHER (NEXT NOTE) (CLIMAX CANTUS-1)]
             [= (+ (#OCCURRENCES (NEXT NOTE) CANTUS-1)
                   (# (SET (NEXT NOTE))))
                1])

    14:68-69 --- [COMPUTE by RULE288, REDUCE by RULE290] --->
        (AND [HIGHER (NEXT NOTE) (CLIMAX CANTUS-1)]
             [= (#OCCURRENCES (NEXT NOTE) CANTUS-1) 0])

    14:70-72 --- [ELABORATE by RULE124, RECOGNIZE by RULE291-292] --->
        (AND [HIGHER (NEXT NOTE) (CLIMAX CANTUS-1)]
             [NOT (IN (NEXT NOTE) CANTUS-1)])

    14:73-79 --- [by analysis:  second conjunct implied by first] --->
        (HIGHER (NEXT NOTE) (CLIMAX CANTUS-1))
```

The form of the solution reflects the original factoring into two cases:

```
    (OR [AND [= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1]
             [LOWER (NEXT NOTE) (CLIMAX CANTUS-1)]]
        [HIGHER (NEXT NOTE) (CLIMAX CANTUS-1)])
```

In this example, condition factoring was used to split the initial expression into two cases. Case 1 was itself split into two branches by factoring a subexpression nested a few levels deep. Propagating the split up one level permitted some evaluation; propagating it up one more level permitted one branch to be reduced to nil. This meant that the factoring condition had to be true for Case 1 to be satisfied. Case 2 also contained a factorable nested subexpression. Here the factoring condition reduced to true without any propagation. This reduced the factored subexpression to one branch, which in turn allowed Case 2 to be simplified considerably.

5.7.3. Conjunctive Subgoaling

Sometimes a problem is solved by splitting it into a conjunction of separately solvable subproblems. In DERIV3 the problem of flushing the Queen is split into two subproblems:

```
(= (LEGALCARDS (QSO)) (SET QS))
3:12 --- [ELABORATE by RULE340] --->
(AND [SUBSET (LEGALCARDS (QSO)) (SET QS)]
     [SUBSET (SET QS) (LEGALCARDS (QSO))])

     (SUBSET (SET QS) (LEGALCARDS (QSO)))
3:13-19 --- [by analysis] --->
     (LEGAL (QSO) QS)
3:20-35 --- [assuming spades are led] --->
     T

     (SUBSET (LEGALCARDS (QSO)) (SET QS))
3:37-97 --- [by analysis] --->
     (UNTIL (PLAYED! QS) (ACHIEVE (LEAD-SPADE ME)))
```

First the subproblem of making the Queen legal for player (QSO) is satisfied by assuming spades are led. Then the subproblem of ensuring that (QSO) has no legal cards besides the Queen is solved by constructing a plan to keep leading spades. The initial problem is split by applying

RULE340: $(= e_1\ e_2)$ -> (and [R $e_1\ e_2$] [R $e_2\ e_1$]),
where R is the standard partial ordering on the domain of e_1 and e_2

Here R = SUBSET, a partial ordering on sets. The "standard" partial orderings include SUBSET for sets, => for propositions, and >= for numbers. The rule is used in DERIV13 with R = >= to split a constraint on the generated tone sequence PROJECT1into a conjunction of two terms:

```
(= (#OCCURRENCES CLIMAX1 PROJECT1) 1)
13:83 --- [ELABORATE by RULE340] --->
(AND [>= (#OCCURRENCES CLIMAX1 PROJECT1) 1]
     [>= 1 (#OCCURRENCES CLIMAX1 PROJECT1)])

     (>= (#OCCURRENCES CLIMAX1 PROJECT1) 1)
13:84-86 --- [by analysis] --->
     (IN CLIMAX1 PROJECT1)

     (>= 1 (#OCCURRENCES CLIMAX1 PROJECT1))
13:87 --- [by RULE153] --->
     (=< (#OCCURRENCES CLIMAX1 PROJECT1) 1)
```

The heuristic search for a sequence satisfying the constraint can be refined by moving the first term to the path-order (§3.5.5.3) and converting the second term into a step-test (§3.5.3.6), while the constraint as first expressed can't be moved earlier in the search.

Both examples exhibit a "divide-and-conquer" approach: although no methods may apply directly to the original problem, each subproblem can be solved by a method specifically appropriate to it.

5.7.4. Making an Initial Split

The first step in separating a problem is to split some part of it into a function of two or more components, such as a conjunction, disjunction, sum, list, or set. Not just any function of more than one argument will do; it must be possible to *propagate* the split far enough to separate the problem into subproblems each of which involves only one of the components.

Usually the initial split is made by expanding the definition of a function used in the problem. This is clear when the definition contains a conjunction, as illustrated in DERIV3:

```
(LEGAL (QSO) QS)
3:20 --- [ELABORATE by RULE124] --->
(AND [HAS (QSO) QS]
     [=> (LEADING (QSO))
         (OR [CAN-LEAD-HEARTS (QSO)]
             [NEQ (SUIT-OF QS) H])]
     [=> (FOLLOWING (QSO))
         (OR [VOID (QSO) (SUIT-LED)]
             [IN-SUIT QS (SUIT-LED)])])
```

Another conjunctive definition is expanded to introduce a split in DERIV13:

```
(LEGAL-CANTUS! PROJECT1)
13:16 --- [ELABORATE by RULE124] --->
(AND [IN (# PROJECT1) (LB:UB 8 16)]
     [FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
        (USABLE-INTERVAL! INTERVAL1)]
     [=< (MELODIC-RANGE PROJECT1) (MAJOR 10)]
     [= (#OCCURRENCES (CLIMAX PROJECT1) PROJECT1) 1])
```

The expanded definition may be based on an ordered list-like construct, as in DERIV6:

```
(TRICK)
6:3 --- [ELABORATE by RULE124] --->
(SCENARIO (EACH P1 (PLAYERS) (PLAY-CARD P1))
          (TAKE-TRICK (TRICK-WINNER)))
```

Or it may be an unordered collection, as in DERIV4:

```
(RANGE LOC)
4:9 --- [ENUMERATE by RULE163] --->
(PARTITION (HANDS (PLAYERS))
           (PILES (PLAYERS))
           (SET DECK POT HOLE))
```

Each example above involves a *homogeneous* function, *i.e.*, its arguments are all of the same type -- propositions, events, or sets. Some *heterogeneous* functions have homogeneous equivalents:

(the x S P_x) is equivalent to "the x | (and [in x S] P_x)"

(set-of x S P_x) is equivalent to {x | (and [in x S] P_x)}

(filter S P) is equivalent to (intersect S {x | P_x})

Splits can be introduced by plugging in definitions of such functions, as illustrated below:

```
(CARDS-IN-SUIT-LED CARDS-PLAYED1)
2:78 --- [ELABORATE by RULE124] --->
(SET-OF C19 CARDS-PLAYED1 (= (SUIT-OF C19) (SUIT-LED)))

(LEGALCARDS (QSO))
3:16 --- [ELABORATE by RULE124] --->
(SET-OF C1 (CARDS) (LEGAL (QSO) C1))

(WINNING-CARD)
5:16 --- [ELABORATE by RULE124] --->
(HIGHEST-IN-SUIT-LED (CARDS-PLAYED))
5:17 --- [ELABORATE by RULE124] --->
(HIGHEST (CARDS-IN-SUIT-LED (CARDS-PLAYED)))
5:18 --- [ELABORATE by RULE124] --->
(THE C2 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
   (FORALL X1 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
      (NOT (HIGHER X1 C))))

(CARDS-IN-SUIT-LED (CARDS-PLAYED))
5:20 --- [ELABORATE by RULE124] --->
(FILTER (CARDS-PLAYED) IN-SUIT-LED)

(TRICK-WINNER)
6:24 --- [ELABORATE by RULE124] --->
(PLAYER-OF (WINNING-CARD))
6:25 --- [ELABORATE by RULE124] --->
(THE P2 (PLAYERS) (= (CARD-OF P2) (WINNING-CARD)))

(CLIMAX PROJECT1)
13:60 --- [ELABORATE by RULE124] --->
(HIGHEST PROJECT1)
13:61 --- [ELABORATE by RULE124] --->
(THE C1 PROJECT1 (FORALL X1 PROJECT1 (NOT (HIGHER X1 C1))))

(#OCCURRENCES (CLIMAX CANTUS-1) (LIST (NEXT NOTE)))
14:19 --- [ELABORATE by RULE124] --->
(# (SET-OF X1 (LIST (NEXT NOTE)) (= X1 (CLIMAX CANTUS-1))))
```

Other rules that introduce splits are listed below with examples of their use.

RULE256: (P ... [f s] ...) -> (and [= (f s) c] [P ... c ...]),
where c is to be selected from (range f) before s is constructed (§2.12.1.1)

```
(ACHIEVE (= (#OCCURRENCES [CLIMAX CANTUS] CANTUS) 1)
13:58   --- [RESTRICT by RULE256] --->
(ACHIEVE (AND [= (CLIMAX CANTUS) CLIMAX1]
              [= (#OCCURRENCES CLIMAX1 CANTUS) 1]))
ASSUMING (SELECT CLIMAX1 (TONES))
```

RULE274: (prefix s (+ i 1)) -> (append [prefix s i] [list (nth s (+ i 1))])

```
(PREFIX CANTUS (+ J 1))
14:12 --- [ELABORATE by RULE124] --->
(APPEND [PREFIX CANTUS J] [LIST (NTH CANTUS (+ J 1))])
```

RULE276: $P_{(\text{next } e)}$ -> (or [and (not (change e)) P_e] [and (change e) $P_{(\text{next } e)}$]) (§5.7.2)

```
(= (#OCCURRENCES [NEXT (CLIMAX CANTUS-1)] (NEXT CANTUS-1)) 1)
14:8 --- [by RULE276] --->
(OR [AND [NOT (CHANGE (CLIMAX CANTUS-1))]
         [= (#OCCURRENCES (CLIMAX CANTUS-1) (NEXT CANTUS-1)) 1]]
    [AND [CHANGE (CLIMAX CANTUS-1)]
         [= (#OCCURRENCES (NEXT (CLIMAX CANTUS-1))
                          (NEXT CANTUS-1))
          1]])
```

RULE338: (HSM with (path-order : (lambda (s) P_s)))
-> (HSM with (step-order : (lambda (i c) $P_{[\text{append } s \text{ (list } c)]}$)))
where $s_{i+1} = c$ (§3.4.3)

```
(HSM1 WITH (PATH-ORDER :
              (LAMBDA (CARDS-PLAYED1)
                 (HAVE-POINTS CARDS-PLAYED1))))
2:101 --- [REFINE by RULE338] --->
HSM1 <- STEP-ORDER :
          (LAMBDA (P1 C21)
             (HAVE-POINTS [APPEND CARDS-PLAYED1 (LIST C21)]))
```

RULE340: (= e_1 e_2) -> (and [R e_1 e_2] [R e_2 e_1]),
where R is the standard partial ordering on the domain of e_1 and e_2 (§5.7.3)

```
(= (LEGALCARDS (QSO)) (SET QS))
3:12 --- [ELABORATE by RULE340] --->
(AND [SUBSET (LEGALCARDS (QSO)) (SET QS)]
     [SUBSET (SET QS) (LEGALCARDS (QSO))])

(= (#OCCURRENCES CLIMAX1 PROJECT1) 1)
13:83 --- [ELABORATE by RULE340] --->
(AND [>= (#OCCURRENCES CLIMAX1 PROJECT1) 1]
     [>= 1 (#OCCURRENCES CLIMAX1 PROJECT1)])
```

RULE344: (undo (= e v)) -> (and [= e v] [become e v'])

```
(UNDO (= (LOC C3) (HAND (QSO)))
3:49 --- [by RULE344] --->
(AND [= (LOC C3) (HAND (QSO))] [BECOME (LOC C3) LOC3])
```

RULE356: P -> (or [was-during (current A) (cause P)] [before (current A) P]) (§2.4.1)

```
(AT QS (PILE P3))
4:47 --- [by RULE356] --->
(OR [WAS-DURING (CURRENT ROUND-IN-PROGRESS)
                (CAUSE (AT QS (PILE P3)))]
    [BEFORE (CURRENT ROUND-IN-PROGRESS) (AT QS (PILE P3))])
```

RULE370: (# (set-of x S Px)) ->
(- [# (set-of x S (before (Current A) Px))]
  [# (set-of x S (was-during (Current A) (undo Px)))]),
provided P cannot become true during events of type A (§2.4.3)

```
(# (SET-OF X1 (CARDS-IN-SUIT S0) (OUT X1)))
10:4-12 --- [by RULE370] --->
(- [# (SET-OF X1 (CARDS-IN-SUIT S0)
          (BEFORE (CURRENT ROUND-IN-PROGRESS) (OUT X1)))]
   [# (SET-OF X1 (CARDS-IN-SUIT S0)
          (WAS-DURING (CURRENT ROUND-IN-PROGRESS)
                      (UNDO (OUT X1))))])
```

RULE380: (= [forall x S_1 P_x] [forall y S_2 R_y]) -> (= R_y (or P_y [in y (diff S_2 S_1)]) (§5.3.4)

```
(= [FORALL I4 (LB:UB 2 (# PROJECT1))
      (USABLE-INTERVAL!
         (INTERVAL (PROJECT1 (- I4 1)) (PROJECT1 I4)))]
   [FORALL I1 (INDICES-OF PROJECT1) Q1])
13:31 --- [REDUCE by RULE380] --->
(= Q1 (OR [USABLE-INTERVAL!
             (INTERVAL (PROJECT1 (- I1 1)) (PROJECT1 I1))]
          [IN I1 (DIFF (INDICES-OF PROJECT1)
                       (LB:UB 2 (# PROJECT1)))]))
```

RULE385: (set-of x S (or ... P ...)) -> (if P S (set-of x S (or ...))) if P is independent of x

```
(SET-OF NOTE1 (TONES)
   (OR [USABLE-INTERVAL!
          (INTERVAL (PROJECT1 (- I1 1)) NOTE1)]
       [= I1 1]))
13:53-54 --- [by RULE385, SIMPLIFY by RULE178] --->
(IF (= I1 1)
    (TONES)
    (SET-OF NOTE1 (TONES)
       (USABLE-INTERVAL!
          (INTERVAL (PROJECT1 (- I1 1)) NOTE1))))
```

RULE389: (HSM with (path-test : (lambda (s) P_s)))
-> (HSM with (step-test : (lambda (i c) $P_{[\text{append s (list c)}]}$)))
where s_{i+1} = c (§3.5.3.6)

```
(HSM1 WITH (PATH-TEST :
              (LAMBDA (PROJECT1)
                 (=< (#OCCURRENCES CLIMAX1 PROJECT1) 1))))
13:105 --- [REFINE by RULE389] --->
HSM1 <- STEP-TEST :
          (LAMBDA (I1 NOTE3)
             (=< (#OCCURRENCES CLIMAX1
                               [APPEND PROJECT1 (LIST NOTE3)])
                 1))
```

These rules introduce splits that can be propagated upward to separate a problem. Several rules for *partial matching* (§5.3.4) are not listed here but have the same effect.

5.7.5. Propagating a Split

Once some component of a problem has been split by one of the means discussed above, the split is *propagated* upward to split the problem into solvable subproblems. FOO's rules for propagating such splits are listed below with examples of their use:

RULE7: (remove-1-from [set-of x S P_x]) -> (undo (and [in x S] P_x))

```
(REMOVE-1-FROM [SET-OF C3 (DIFF (CARDS) (SET QS))
                          (LEGAL (QSO) C3)])
3:41 --- [REDUCE by RULE7] --->
(UNDO (AND [IN C3 (DIFF (CARDS) (SET QS))] [LEGAL (QSO) C3]))

(REMOVE-1-FROM [SET-OF C1 (CARDS-IN-HAND ME) (IN-SUIT C1 SO)])
8:4 --- [REDUCE by RULE7] --->
(UNDO (AND [IN C1 (CARDS-IN-HAND ME)] [IN-SUIT C1 SO]))
```

RULE10: (in c [set-of x S P_x]) -> (and [in c S] P_c)

```
(IN (CARDS-PLAYED1 ME) [SET-OF C22 CARDS-PLAYED1
                             (= (SUIT-OF C22) (SUIT-LED))])
2:91 --- [REMOVE-QUANT by RULE10] --->
(AND [IN (CARDS-PLAYED1 ME) CARDS-PLAYED1]
     [= (SUIT-OF (CARDS-PLAYED1 ME)) (SUIT-LED)])

(IN QS [SET-OF C1 (CARDS) (LEGAL (QSO) C1)])
3:17 --- [REMOVE-QUANT by RULE10] --->
(AND [IN QS (CARDS)] [LEGAL (QSO) QS])

(IN C3 [SET-OF C4 (CARDS) (HAS-POINTS C4)])
6:16 --- [REMOVE-QUANT by RULE10] --->
(AND [IN C3 (CARDS)] [HAS-POINTS C3])

(IN C1 [SET-OF C2 (CARDS) (HAS ME C2)])
8:7 --- [REMOVE-QUANT by RULE10] --->
(AND [IN C1 (CARDS)] [HAS ME C1])
```

RULE35: (affect (and ... P ...)) -> (and [affect P] ...)

```
(UNDO (AND [IN C3 (DIFF (CARDS) (SET QS))] [LEGAL (QSO) C3]))
3:42 --- [REDUCE by RULE35] --->
(AND [UNDO (LEGAL (QSO) C3)] [IN C3 (DIFF (CARDS) (SET QS))])
```

```
(UNDO (AND [HAS (QSO) C3]
           [=> (LEADING (QSO))
               (OR [CAN-LEAD-HEARTS (QSO)]
                   [NEQ (SUIT-OF C3) H])]
           [=> (FOLLOWING (QSO))
               (OR [VOID (QSO) (SUIT-LED)]
                   [IN-SUIT C3 (SUIT-LED)])]))
3:44 --- [REDUCE by RULE35] --->
(AND [UNDO (HAS (QSO) C3)]
     [=> (LEADING (QSO))
         (OR [CAN-LEAD-HEARTS (QSO)]
             [NEQ (SUIT-OF C3) H])]
     [=> (FOLLOWING (QSO))
         (OR [VOID (QSO) (SUIT-LED)]
             [IN-SUIT C3 (SUIT-LED)])])

(UNDO (AND [IN C1 (CARDS-IN-HAND ME)] [IN-SUIT C1 S0]))
8:5 --- [REDUCE by RULE35] --->
(AND [UNDO (IN C1 (CARDS-IN-HAND ME))] [IN-SUIT C1 S0])

(UNDO (AND [IN C1 (CARDS)] [HAS ME C1]))
8:8 --- [REDUCE by RULE35] --->
(AND [UNDO (HAS ME C1)] [IN-SUIT C1 S0])

(CAUSE (AND [HAS P0 C1] [IN-SUIT C1 S0]))
12:7 --- [REDUCE by RULE35] --->
(AND [CAUSE (HAS P0 C1)] [IN-SUIT C1 S0])
```

RULE64: (= c (the x S P_x)) -> (and [in c S] P_c)

```
(= (CARD-OF ME)
   (THE C2 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
      (FORALL X1 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
         (NOT (HIGHER X1 C2)))))
5:19 --- [by RULE64] --->
(AND [IN (CARD-OF ME)
         (CARDS-IN-SUIT-LED (CARDS-PLAYED))]
     [FORALL X1 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
        (NOT (HIGHER X1 (CARD-OF ME)))])
```

RULE131: (= (the x S P_x) c) -> (and [in c S] P_c)

```
(= (THE P2 (PLAYERS) (= (CARD-OF P2) (WINNING-CARD))) ME)
6:26 --- [REMOVE-QUANT by RULE131] --->
(AND [IN ME (PLAYERS)] [= (CARD-OF ME) (WINNING-CARD)])

(= (THE C1 PROJECT1
      (FORALL X1 PROJECT1 (NOT (HIGHER X1 C1))))
   CLIMAX1)
13:62 --- [REMOVE-QUANT by RULE131] --->
(AND [IN CLIMAX1 PROJECT1]
     [FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1))])
```

RULE135: (Q x [set-of y S P_y] R_x) -> (Q x S (and P_x R_x))

```
(EXISTS C11 [SET-OF C12 (CARDS) (HAS P5 C12)]
   (IN-SUIT C11 (SUIT-LED)))
2:45 --- [by RULE135] --->
(EXISTS C11 (CARDS) (AND [HAS P5 C11] [IN-SUIT C11 (SUIT-LED)]))
```

RULE149: (in c (f S)) -> (and P_c [in c S])) if f is defined as (lambda (S) (filter S P))

```
(IN (CARD-OF ME) (CARDS-IN-SUIT-LED (CARDS-PLAYED)))
6:33 --- [by RULE149] --->
(AND [IN-SUIT-LED (CARD-OF ME)] [IN (CARD-OF ME) (CARDS-PLAYED)])

(IN CARD1 (CARDS-IN-SUIT-LED (CARDS-PLAYED)))
6:45 --- [by RULE149] --->
(AND [IN-SUIT-LED CARD1] [IN CARD1 (CARDS-PLAYED)])
```

RULE182: (in e (set e_1 ... e_n)) -> (or [= e e_1] ... [= e e_n])

```
(IN (LOC QS) (SET DECK POT HOLE (HAND ME)))
4:32 --- [DISTRIBUTE by RULE182] --->
(OR [= (LOC QS) DECK]
    [= (LOC QS) POT]
    [= (LOC QS) HOLE]
    [= (LOC QS) (HAND ME)])

(IN I1 (SET 1))
13:36 --- [DISTRIBUTE by RULE182] --->
(OR [= I1 1])
```

RULE284: (f ... [g e_1 ... e_n] ...) -> (g' [f ... e_1 ...] ... [f ... e_n ...])
if (lambda (x) (f ... x ...)) is a homomorphism,
where g, g' correspond to +, +' in the homomorphism condition $f(x+y) = f(x) +' f(y)$

```
(CHOICE-SEQ-OF [SCENARIO (EACH P1 (PLAYERS) (PLAY-CARD P1))
                         (TAKE-TRICK (TRICK-WINNER))])
2:6 --- [by RULE284] --->
(APPEND [CHOICE-SEQ-OF (EACH P1 (PLAYERS) (PLAY-CARD P1))]
        [CHOICE-SEQ-OF (TAKE-TRICK (TRICK-WINNER))])

(HAVE-POINTS [APPEND CARDS-PLAYED1 (LIST C21)])
2:102 --- [DISTRIBUTE by RULE284] --->
(OR [HAVE-POINTS CARDS-PLAYED1] [HAVE-POINTS (LIST C21)])

(IN (HAND ME) [UNION (HANDS (PLAYERS))
                     (PILES (PLAYERS))
                     (SET DECK POT HOLE)])
4:21 --- [DISTRIBUTE by RULE284] --->
(OR [IN (HAND ME) (HANDS (PLAYERS))]
    [IN (HAND ME) (PILES (PLAYERS))]
    [IN (HAND ME) (SET DECK POT HOLE)])

(IN (LOC QS)
    [UNION (SET DECK POT HOLE (HAND ME)) (PILES (PLAYERS))])
4:31 --- [DISTRIBUTE by RULE284] --->
(OR [IN (LOC QS) (SET DECK POT HOLE (HAND ME))]
    [IN (LOC QS) (PILES (PLAYERS))])

(DURING [SCENARIO (EACH P1 (PLAYERS) (PLAY-CARD P1))
                  (TAKE-TRICK (TRICK-WINNER))]
        (TAKE ME QS))
5:7 --- [DISTRIBUTE by RULE284] --->
(OR [DURING (EACH P1 (PLAYERS) (PLAY-CARD P1)) (TAKE ME QS)]
    [DURING (TAKE-TRICK (TRICK-WINNER)) (TAKE ME QS)])
```

```
(DURING [SCENARIO (EACH P1 (PLAYERS) (PLAY-CARD P1)
                  (TAKE-TRICK (TRICK-WINNER))]
        (TAKE-POINTS ME))
6:4 --- [DISTRIBUTE by RULE284] --->
(OR [DURING (EACH P1 (PLAYERS) (PLAY-CARD P1)) (TAKE-POINTS ME)]
    [DURING (TAKE-TRICK (TRICK-WINNER)) (TAKE-POINTS ME)])

(#OCCURRENCES CLIMAX1 [APPEND PROJECT1 (LIST NOTE3)])
13:106 --- [DISTRIBUTE by RULE284] --->
(+ [#OCCURRENCES CLIMAX1 PROJECT1]
   [#OCCURRENCES CLIMAX1 (LIST NOTE3)])

(#OCCURRENCES (CLIMAX CANTUS-1)
              [APPEND CANTUS-1 (LIST (NEXT NOTE))])
14:18 --- [DISTRIBUTE by RULE284] --->
(+ [#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1]
   [#OCCURRENCES (CLIMAX CANTUS-1) (LIST (NEXT NOTE))])

(#OCCURRENCES (NEXT NOTE)
              [APPEND CANTUS-1 (LIST (NEXT NOTE))])
14:62 --- [DISTRIBUTE by RULE284] --->
(+ [#OCCURRENCES (NEXT NOTE) CANTUS-1]
   [#OCCURRENCES (NEXT NOTE) (LIST (NEXT NOTE))])
```

RULE287: (... [if P e e'] ...) -> (if P [... e ...] [... e' ...])

```
(# [IF (= (NEXT NOTE) (CLIMAX CANTUS-1)) (SET (NEXT NOTE)) NIL])
14:22 --- [DISTRIBUTE by RULE287] --->
(IF (= (NEXT NOTE) (CLIMAX CANTUS-1))
    [# (SET (NEXT NOTE)))
    [# NIL))

(= (+ (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1)
      [IF (= (NEXT NOTE) (CLIMAX CANTUS-1)) 1 0])
   1)
14:25 --- [DISTRIBUTE by RULE287] --->
(IF (= (NEXT NOTE) (CLIMAX CANTUS-1))
    [= (+ (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1) 1]
    [= (+ (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 0) 1])
```

RULE321: (=> [P ...] [and ... [P ...] ...]) -> (and [=> (P ...) (P ...)] ...)

```
(=> [HAS P1 C2) (AND [HAS P5 C11] [IN-SUIT C11 (SUIT-LED)]))
2:47 --- [REDUCE by RULE321] --->
(AND [=> (HAS P1 C2) (HAS P5 C11)] [IN-SUIT C11 (SUIT-LED)])
```

RULE361: (in c (filter S P)) -> (and [in c S] P_c)

```
(IN (CARD-OF ME) (FILTER (CARDS-PLAYED) IN-SUIT-LED))
5:21 --- [ELABORATE by RULE361] --->
(AND [IN (CARD-OF ME) (CARDS-PLAYED)] [IN-SUIT-LED (CARD-OF ME)])

(IN QS (FILTER (CARDS-PLAYED) IN-SUIT-LED))
5:41 --- [ELABORATE by RULE361] --->
(AND [IN QS (CARDS-PLAYED)] [IN-SUIT-LED QS])
```

RULE371: (before A (P e_1 ... e_n)) -> (P (before A e_1) ... (before A e_n))

```
(BEFORE (CURRENT ROUND-IN-PROGRESS)
        (NOT (OR [IN-POT X1] [AT X1 HOLE] [HAS ME X1])))
10:15 --- [DISTRIBUTE by RULE371] --->
(NOT (BEFORE (CURRENT ROUND-IN-PROGRESS)
             (OR [IN-POT X1] [AT X1 HOLE] [HAS ME X1])))

10:16 --- [DISTRIBUTE by RULE371] --->
(NOT (OR [BEFORE (CURRENT ROUND-IN-PROGRESS)
                 (IN-POT X1)]
         [BEFORE (CURRENT ROUND-IN-PROGRESS)
                 (AT X1 HOLE)]
         [BEFORE (CURRENT ROUND-IN-PROGRESS)
                 (HAS ME X1)]))
```

RULE375: (# [set-of x S (not P_x)]) -> (- [# S] [# (set-of x S P_x)])

```
(# [SET-OF X1 (CARDS-IN-SUIT S0)
      (NOT (BEFORE (CURRENT ROUND-IN-PROGRESS) (HAS ME X1)))])
10:23   --- [by RULE375] --->
(- [# (CARDS-IN-SUIT S0)]
   [# (SET-OF X1 (CARDS-IN-SUIT S0)
        (BEFORE (CURRENT ROUND-IN-PROGRESS) (HAS ME X1)))])
```

RULE393: (lambda (t) [f ... e_t ...]) -> (f ... [lambda (t) e_t] ...)

```
(LAMBDA (J) [APPEND (PREFIX CANTUS J) (LIST (NTH CANTUS (+ J 1)))])
14:13 --- [by RULE393] --->
(APPEND [LAMBDA (J) (PREFIX CANTUS J)]
        [LAMBDA (J) (LIST (NTH CANTUS (+ J 1)))])

(LAMBDA (J) [LIST (NOTE (+ J 1))])
14:16 --- [by RULE393] --->
(LIST [LAMBDA (J) (NOTE (+ J 1))])
```

5.7.6. Problem Merging

When two or more subproblems share some regularity that allows them to be solved by a single method, it is sometimes useful to *merge* them into a single problem. For example, in DERIV14, two separately derived constraints are merged in terms of a single known predicate:

```
(NOT (OR [HIGHER (NEXT NOTE) (CLIMAX CANTUS-1)]
         [= (NEXT NOTE) (CLIMAX CANTUS-1)]))
14:48 --- [COLLECT by RULE297] --->
(NOT ([OR HIGHER =] (NEXT NOTE) (CLIMAX CANTUS-1)))
14:49 --- [by RULE298] --->
([NOT (OR HIGHER =)] (NEXT NOTE) (CLIMAX CANTUS-1))
14:50 --- [RECOGNIZE by RULE299] --->
(LOWER (NEXT NOTE) (CLIMAX CANTUS-1))
```

The merging transformation at step 14:48 is performed by

RULE297: (B [P e_1 e_2] [P' e_1 e_2]) -> ([B P P'] e_1 e_2)

Note that RULE297 only applies to propositions sharing the same respective arguments e_1 and e_2.

In DERIV5, two pieces of advice are merged into a single goal:

```
(UNTIL (PLAYED! QS)
       (AND [ACHIEVE (LEAD-SPADE ME)]
            [ACHIEVE (NOT (DURING (TRICK) (TAKE ME QS)))]))
5:4 --- [COLLECT by RULE360] --->
(UNTIL (PLAYED! QS)
       (ACHIEVE (AND [LEAD-SPADE ME]
                     [NOT (DURING (TRICK) (TAKE ME QS))])))
```

The second conjunct is ultimately transformed into the condition that player ME's card be lower than QS, and this condition is used to modify the "lead spades" plan to "lead *safe* spades":

```
(UNTIL (PLAYED! QS)
       (ACHIEVE (AND [LEADING ME]
                     [SPADE! (CARD-OF ME)]
                     [HIGHER QS (CARD-OF ME)])))
5:53 --- [RECOGNIZE by RULE123] --->
(UNTIL (PLAYED! QS) (ACHIEVE (LEAD-SAFE-SPADE ME)))
```

The problem-merging transformation at step 5:4 is performed by

RULE360: (and [achieve P_1] ... [achieve P_n]) -> (achieve (and P_1 ... P_n))

Note that the merged subproblems are both of the form (ACHIEVE ...).

These are the only examples of problem merging in the derivations; evidently problem separation is a much more common strategy in operationalization. Indeed, problem merging is only applicable when the problems to be merged share some regularity that allows them to be solved by a single method. In contrast, problem separation is useful whenever splitting a problem into subproblems permits each one to be solved by the method most appropriate to it. It is unsurprising that the latter situation occurs more often; two or more randomly generated subproblems are unlikely to be solvable by the same method.

Actually, subproblems need not be solvable *individually* by a common method in order to be merged; the only requirement is that they fit together into a *combination* that is solvable as a unit. One can construct cases where such a combination is easier to solve than the individual subproblems. For example, it may be difficult to achieve either of the goals C and ~C, but it is trivial to achieve (or C ~C)! Similarly, it may be difficult to evaluate e and -e separately, but it is easy to evaluate (+ e -e).

5.7.7. Problem Separation: Summary

The idea of separating a problem into individually solvable subproblems characterizes case analysis, condition factoring, partial matching, and conjunctive subgoaling. The key steps in problem separation are *splitting* a problem component and *propagating* this split upward in the problem description. Finding the initial split usually involves bringing in domain knowledge, typically by expanding the definition of a term used in the problem. Propagating a split upward uses general rules, and produces subproblems containing successively more context. This process continues until each subproblem can be solved by available methods. Sometimes it is not necessary to solve all the subproblems, for example when achieving a disjunction or disproving a conjunction. Problem separation is a *general technique for applying special-purpose methods*, since it allows each subproblem to be solved using the methods appropriate to it, independently of how the other subproblems are solved. This flexibility appears to explain the contrasting rarity of *problem merging*, which only works when the merged problems are sufficiently similar to be solved by a common method.

5.8. Translation

A useful strategy for making a problem easier to solve is

Translate the problem into the language of known methods.

While many rules transform one expression into another that contains different symbols, this section focuses specifically on rules whose *primary purpose* is translation rather some other process (*e.g.*, evaluation, partial matching, problem separation). Translation is one kind of *change of representation* [Korf 80]. A translation rule has the general form (f e_1 ... e_n) -> (g e_1' ... e_k') and introduces a *change of vocabulary*, as opposed to a *change of syntax* (§5.10). It can be classified according to which of the concepts f and g is more specific; f is considered *more specific* than g if every expression (f ...) is expressible as (g ...), but not *vice versa*. This criterion splits the class of translation rules into three kinds:

1. *Elaboration*: f is more specific than g. Expanding the definition of f is elaboration.
2. *Recognition*: g is more specific than f. Substituting g for its definition is recognition.
3. *Lateral rephrasing*: neither f nor g is more specific. Translation other than elaboration or recognition is lateral rephrasing.

The rest of this section is organized as follows. (§5.8.1) illustrates the concept of translation by means of an example. The succeeding three sections present FOO's rules for elaboration (§5.8.2), recognition (§5.8.3), and lateral rephrasing (§5.8.4). Recognition and elaboration correspond to Darlington and Burstall's "folding" and "unfolding" operations [Darlington 76]. (§5.8.5) characterizes the three kinds of translation rules as operators moving in different directions in the operationalization problem space.

5.8.1. An Example of Translation from Numerical to Set Concepts

A nice example of translation occurs in DERIV14 in the course of the reduction

```
(= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 0) -> NIL
```

First a sub-expression is *elaborated* by plugging in the definition of #OCCURRENCES:

```
   (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1)
14:29 --- [ELABORATE by RULE124] --->
   (# [SET-OF X2 CANTUS-1 (= X2 (CLIMAX CANTUS-1))])
```

This produces an expression of the form (= (# [set-of x S P_x]) 0), which can be *recognized* as (empty [set-of x S P_x]). Such a transformation qualifies as recognition since any expression of the form (empty S) can be expressed as (= (# S) 0), but not every expression (= e e') can be expressed in the form (empty ...) -- at least in a "natural" way. The latter qualification reflects the difficulty of precisely defining relative specificity so as to rule out perverse equivalents for (= e e') such as (empty [set-of x {e} (≠ x e')]). Perhaps f should be called more specific than g if (f ...) can be expressed in terms of g *more easily* than *vice versa* -- whatever "more easily" means.

The next step in the derivation reformulates an equality as a negated quantification:

```
(= (# [SET-OF X2 CANTUS-1 (= X2 (CLIMAX CANTUS-1))]) 0)
14:30 --- [RECOGNIZE by RULE291] --->
(NOT (EXISTS X2 CANTUS-1 (= X2 (CLIMAX CANTUS-1))))
```

This step combines *recognition* and *lateral rephrasing*:

```
(= (# [SET-OF X2 CANTUS-1 (= X2 (CLIMAX CANTUS-1))]) 0)
 --- [recognize] --->
(EMPTY [SET-OF X2 CANTUS-1 (= X2 (CLIMAX CANTUS-1))])
 --- [laterally rephrase] --->
(NOT (EXISTS X2 CANTUS-1 (= X2 (CLIMAX CANTUS-1))))
```

The rule for this composite transformation is

RULE291: (= (# [set-of x S P_x]) 0) -> (not (exists x S P_x))

An equality quantified over a set can be *recognized* in terms of membership in that set:

```
          (EXISTS X2 CANTUS-1 (= X2 (CLIMAX CANTUS-1)))
     14:31 --- [RECOGNIZE by RULE292] --->
          (IN (CLIMAX CANTUS-1) CANTUS-1)
```

This transformation is expressed by

RULE292: (exists x S (= x e)) -> (in e S)

Now the sub-expression `(CLIMAX CANTUS-1)` is *elaborated* by plugging in its definition:

```
              (CLIMAX CANTUS-1)
     14:32    --- [ELABORATE by RULE124] --->
              (HIGHEST CANTUS-1)
```

The latter expression fits the language of an opportunistic evaluation method (§5.2.3):

RULE293: (in (one-of S) S) -> T

This method helps reduce the overall expression to `NIL`:

```
     (NOT (IN (HIGHEST CANTUS-1) CANTUS-1))
     14:33 --- [SIMPLIFY by RULE293] --->
     (NOT T)
     14:34 --- [COMPUTE by RULE236] --->
     NIL
```

Most of this example consisted of translating the initial expression from the language of *numerical* concepts, like `#` and `0`, into the language of *set-related* concepts, like `IN` and `HIGHEST`. Once this translation was accomplished, by a combination of elaboration, recognition, and lateral rephrasing, the desired proof was obtained immediately by applying a method expressed in the language of sets.

Note that using translation to solve this problem in an automatic (rather than user-guided) mode would require capabilities lacking in FOO (§8.1.1):

1. A way to make the initial decision to restate the problem in terms of sets. What clue in the initial problem suggests this as a plausible goal? One possibility is to establish a permanent (*i.e.*, problem-independent) goal of applying powerful problem-solving and problem-reducing methods, *e.g.*, opportunistic evaluation rules like RULE293 in the example. The desire to apply RULE293 would motivate the attempt to translate the initial problem into the language of sets. A potential difficulty of this approach is an excessive number of such goals. Avoiding this might require an efficient means to eliminate infeasible methods, *e.g.*, by some kind of intersection search between the concepts used in the problem and the concepts used in the method.
2. A way to guide the translation process toward a chosen goal without wasting too many moves. One approach is to use means-end analysis, with rules as operators (§7).

The preceding example is almost pure translation, and therefore ideal for illustrative purposes. More typically, translation is interleaved with other transformations, as in DERIV4, where the problem of deciding whether the Queen of spades is out is translated into the language of the pigeon-hole principle. The translation begins by successive elaboration:

```
(QS-OUT)
4:1 --- [ELABORATE by RULE124] --->
(OUT QS)

4:2 --- [ELABORATE by RULE124] --->
(EXISTS P1 (OPPONENTS ME) (HAS P1 QS))

                               (HAS P1 QS)
4:3                            --- [ELABORATE by RULE124] --->
                               (AT QS (HAND P1))

4:4                            --- [ELABORATE by RULE124] --->
                               (= (LOC QS) (HAND P1))
```

This is followed by a change of variable using

RULE161: (Q x S [P ... (f x) ...]) -> (Q y (project f S) [P ... y ...])

The effect is to *restructure* the quantification (§5.10):

```
(EXISTS P1 (OPPONENTS ME) [= (LOC QS) (HAND P1)])
4:5 --- [COLLECT by RULE161] --->
(EXISTS Y1 (PROJECT HAND (OPPONENTS ME)) [= (LOC QS) Y1])
```

The equality, quantified over hands instead of players, is recognized in terms of set membership:

```
4:6 --- [RECOGNIZE by RULE162] --->
(IN (LOC QS) (PROJECT HAND (OPPONENTS ME)))
```

The recognition is performed by

RULE162: (exists x S (= e x)) -> (in e S)

The set membership matches the problem statement for the pigeon-hole principle (§2.2):

```
(IN (LOC QS) (PROJECT HAND (OPPONENTS ME)))
4:7 --- [by RULE169] --->
(NOT (IN (LOC QS) (DIFF (RANGE LOC)
                        (PROJECT HAND (OPPONENTS ME)))))
```

A *restructuring* rule is used in this example to help the translation go through. In general, translation is interspersed with evaluation, simplification, reduction, and other analysis methods.

5.8.2. Elaboration

Elaboration unpacks implicit details to make them available for analysis, moving *downward* from compact descriptions based on highly specific concepts to extensive descriptions based on more primitive terms at a lower level of detail. Elaborating a specialized concept is useful when available methods do not apply to it directly, but do apply to more general concepts used in its expanded form. In this sense, elaboration provides a way to exploit general methods.

The most common kind of elaboration consists of plugging in the definition of a concept using

RULE124: (f e_1 ... e_n) -> e, where f is defined as (lambda (x_1 ... x_n) <body>) and e is the result of substituting e_1 ... e_n for x_1 ... x_n throughout <body>

For example:

```
(AVOID (TAKE-POINTS ME) (TRICK))
6:2 --- [ELABORATE by RULE124] --->
(ACHIEVE (NOT (DURING (TRICK) (TAKE-POINTS ME))))
```

Here f = `AVOID`, defined as `(LAMBDA (E S) (ACHIEVE (NOT (DURING S E))))`.

RULE124 is by far the most frequently used of all the rules in FOO; it accounts for more than 100 steps in the derivations.

FOO also has some special-case elaboration rules, listed below with examples of their use.

RULE4: (subset S_1 S_2) -> (empty (diff S_1 S_2))

```
(SUBSET (LEGALCARDS (QSO)) (SET QS))
3:37  --- [by RULE4] --->
(EMPTY (DIFF (LEGALCARDS (QSO)) (SET QS)))
```

RULE14: (in e (project f S)) -> (exists x S (= e (f x)))

```
(IN (LOC QS) (PROJECT PILE (PLAYERS)))
4:45 --- [by RULE14] --->
(EXISTS P3 (PLAYERS) (= (LOC QS) (PILE P3)))
```

RULE14 collapses an elaborate-and-transfer (§5.10.2) sequence:

(in e (project f S)) ->
(exists y (project f S) (= e y)) ->
(exists x S (= e (f x)))

RULE15: (partition S_1 ... S_n) -> (union S_1 ... S_n), recording assumption (disjoint S_1 ... S_n)

```
(PARTITION (HANDS (PLAYERS))
           (PILES (PLAYERS))
           (SET DECK POT HOLE))
4:10 --- [GENERALIZE by RULE15] --->
(UNION (HANDS (PLAYERS))
       (PILES (PLAYERS))
       (SET DECK POT HOLE))
ASSUME (DISJOINT (HANDS (PLAYERS))
                 (PILES (PLAYERS))
                 (SET DECK POT HOLE))
```

RULE381: (indices-of S) -> (lb:ub 1 (# S)), where S is a sequence indexed by integer

```
(INDICES-OF PROJECT1)
13:32 --- [ELABORATE by RULE381] --->
(LB:UB 1 (# PROJECT1))
```

5.8.3. Recognizing Known Concepts

Recognition moves *upward* from primitive concepts to specialized ones that may yield to special-case methods; thus recognition exploits special cases. This was illustrated in an earlier example: although FOO has no direct evaluation rules for conditions of the form (= (# ...) k) for arbitrary k, a condition of the form (= (# ...) 0) was evaluated by using knowledge about set membership.

The most common kind of recognition consists of substituting a concept for its definition using

RULE123: e -> (f e_1 ... e_n) if f is defined as (lambda (x_1 ... x_n) <body>)
and e is the result of substituting e_1 ... e_n for x_1 ... x_n throughout <body>

This rule was the second most frequently used (over 40 times). It is illustrated in DERIV8:

```
(MOVE C1 (HAND ME) LOC1)
8:11 --- [RECOGNIZE by RULE123, binding LOC1 <- POT] --->
(PLAY ME C1)
```

A variant form was used in DERIV14 to recognize lambda expressions as known functions:

RULE267: (lambda (x_1 ... x_n) e) -> f if f is defined as (lambda (x_1' ... x_n') <body>)
and e is the result of substituting x_1 ... x_n for x_1' ... x_n' throughout <body>

```
(LAMBDA (J1) (PREFIX CANTUS J1))
14:3 --- [RECOGNIZE by RULE267] --->
CANTUS-1
```

FOO's other recognition rules are listed below with examples of their use.

RULE162: (exists x S (= e x)) -> (in e S)

```
(EXISTS Y1 (PROJECT HAND (OPPONENTS ME)) (= (LOC QS) Y1))
4:6 --- [REMOVE-QUANT by RULE162] --->
(IN (LOC QS) (PROJECT HAND (OPPONENTS ME)))

(EXISTS P1 (OPPONENTS ME) (= P0 P1))
9:36 --- [REMOVE-QUANT by RULE162] --->
(IN P0 (OPPONENTS ME))
```

RULE291: (= (# [set-of x S P_x]) 0) -> (not (exists x S P_x))

```
(= (# (SET-OF X5 PROJECT1 (= X5 CLIMAX1))) 0)
13:113 --- [RECOGNIZE by RULE291] --->
(NOT (EXISTS X5 PROJECT1 (= X5 CLIMAX1)))

(= (# (SET-OF X2 CANTUS-1 (= X2 (CLIMAX CANTUS-1)))) 0)
14:30 --- [RECOGNIZE by RULE291] --->
(NOT (EXISTS X2 CANTUS-1 (= X2 (CLIMAX CANTUS-1))))
```

RULE291 collapses a recognize-and-rephrase sequence:

(= (# [set-of x S Px]) 0) ->
(empty [set-of x S Px]) ->
(not (exists x S Px))

RULE292: (exists x S (= x e)) -> (in e S) -- *symmetric form of RULE162*

```
(EXISTS X2 PROJECT1 (= X2 CLIMAX1))
13:86 --- [RECOGNIZE by RULE292] --->
(IN CLIMAX1 PROJECT1)

(EXISTS X2 CANTUS-1 (= X2 (CLIMAX CANTUS-1)))
14:31 --- [RECOGNIZE by RULE292] --->
(IN (CLIMAX CANTUS-1) CANTUS-1)

(EXISTS X5 CANTUS-1 (= X5 (NEXT NOTE)))
14:72 --- [RECOGNIZE by RULE292] --->
(IN (NEXT NOTE) CANTUS-1)
```

RULE294: (if C nil P) -> (and P [not C])

```
(IF (= (NEXT NOTE) (CLIMAX CANTUS-1))
    NIL
    (= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1))
14:36 --- [RECOGNIZE by RULE294] --->
(AND [= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1]
     [NOT (= (NEXT NOTE) (CLIMAX CANTUS-1))])
```

RULE352: (= P nil) -> (not P)

```
(= (LEADING (QS0)) NIL)
3:89 --- [SIMPLIFY by RULE352] --->
(NOT (LEADING (QS0)))
```

RULE358: (cause (at obj loc)) -> (move obj loc' loc), asserting variable loc' ≠ loc

```
(CAUSE (AT QS (PILE P3)))
4:51 --- [RECOGNIZE by RULE358] --->
(MOVE QS LOC1 (PILE P3))

(CAUSE (AT X1 (HAND P1)))
10:8 --- [RECOGNIZE by RULE358] --->
(MOVE X1 LOC1 (HAND P1))
```

RULE367: (not (P e)) -> (P' e), where P' is the opposite of P

```
(NOT (LOW (CARD-OF ME)))
7:8 --- [by RULE367] --->
(HIGH (CARD-OF ME))
```

RULE368: (undo (at obj loc)) -> (move obj loc loc'), asserting variable loc' ≠ loc

```
(UNDO (AT C1 (HAND ME)))
8:10 --- [RECOGNIZE by RULE368] --->
(MOVE C1 (HAND ME) LOC1)
```

RULE377: (undo (not P)) -> (cause P)

```
(UNDO (NOT (EXISTS C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0))))
12:3 --- [RECOGNIZE by RULE377] --->
(CAUSE (EXISTS C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0)))
```

RULE382: (diff [lb:ub l_1 u] [lb:ub l_2 u]) -> (lb:ub l_1 (- l_2 1))

```
(DIFF [LB:UB 1 (# PROJECT1)] [LB:UB 2 (# PROJECT1)])
13:33 --- [REDUCE by RULE382] --->
(LB:UB 1 (- 2 1))
```

RULE387: (>= (# [set-of x S P_x]) 1) -> (exists x S P_x)

```
(>= (# (SET-OF X2 PROJECT1 (= X2 CLIMAX1))) 1)
13:85 --- [RECOGNIZE by RULE387] --->
(EXISTS X2 PROJECT1 (= X2 CLIMAX1))
```

RULE387 is closely related to RULE391:

(>= (# [set-of x S P_x]) 1)
-> (not (< (# [set-of x S P_x]) 1)) -- *"greater or equal" = "not less than"*
-> (not (=< (# [set-of x S P_x]) 0)) -- *"less than 1" = "0 or less" for integers*
-> (not (= (# [set-of x S P_x]) 0)) -- *"non-positive" = "0" for non-negative numbers*
-> (not (empty [set-of x S P_x])) -- *recognize "cardinality 0" as "empty"*
-> (exists x S P_x) -- *recognize "non-empty intension" as existential predicate*

RULE391: (if C P T) -> (or [not C] P)

```
(IF (= CLIMAX1 NOTE3)
    (=< (#OCCURRENCES CLIMAX1 PROJECT1) 0)
    T)
13:109 --- [RECOGNIZE by RULE391] --->
(OR [NOT (= CLIMAX1 NOTE3)]
    [=< (#OCCURRENCES CLIMAX1 PROJECT1) 0])
```

RULE392: $(=\!<\ e\ 0) \rightarrow (=\ e\ 0)$ if e is non-negative (*e.g.*, the size of a set)

```
(=< (#OCCURRENCES CLIMAX1 PROJECT1) 0)
13:111 --- [RECOGNIZE by RULE392] --->
(= (#OCCURRENCES CLIMAX1 PROJECT1) 0)
```

5.8.4. Rephrasing in Different Terms at the Same Level of Detail

Lateral rephrasing translates a problem into different terminology so as to allow application of methods expressed in it, moving *sideways* at more or less the same level of conceptual detail. It covers translation other than elaboration and recognition, and provides a way to exploit methods expressed in different terms than the problem at hand.

FOO's lateral rephrasing rules are listed below with examples of their use.

RULE153: $(R\ e_1\ e_2) \rightarrow (R^T\ e_2\ e_1)$, where R^T is the transpose of the binary relation R

```
(HIGHER X1 (CARD-OF ME))
7:5 --- [by RULE153] --->
(LOWER (CARD-OF ME) X1)

(HIGHER (FIND-ELT (CARD-IN-SUIT-LED (CARDS-PLAYED))) (CARD-OF ME))
5:42 --- [by RULE153] --->
(LOWER (CARD-OF ME) (FIND-ELT (CARD-IN-SUIT-LED (CARDS-PLAYED))))

(HIGHER DK (CARD-OF ME))
6:62 --- [by RULE153] --->
(LOWER (CARD-OF ME) DK)

(>= 1 (#OCCURRENCES CLIMAX1 PROJECT1))
13:87 --- [by RULE153] --->
(=< (#OCCURRENCES CLIMAX1 PROJECT1) 1)
```

RULE159: (not (exists x S P_x)) -> (empty (set-of x S P_x))

```
(NOT (EXISTS C1 (CARDS-IN-HAND ME) (IN-SUIT C1 S0)))
8:2 --- [by RULE159] --->
(EMPTY (SET-OF C1 (CARD-IN-HAND ME) (IN-SUIT C1 S0)))
```

RULE164: (diff S_1 [diff S_2 S_3]) -> (union [diff S_1 S_2] [intersect S_1 S_3])

```
(DIFF (UNION (HANDS (PLAYERS))
             (PILES (PLAYERS))
             (SET DECK POT HOLE))
      [DIFF (HANDS (PLAYERS)) (SET (HAND ME))])
4:15 --- [DISTRIBUTE by RULE164] --->
(UNION [DIFF (UNION (HANDS (PLAYERS))
                    (PILES (PLAYERS))
                    (SET DECK POT HOLE))
             (HANDS (PLAYERS)))
       [INTERSECT (UNION (HANDS (PLAYERS))
                         (PILES (PLAYERS))
                         (SET DECK POT HOLE)]
                  (SET (HAND ME))))
```

RULE164 expresses a set identity derivable by expanding the definition of set difference, applying DeMorgan's Law, and recognizing the definition of set difference. The following proof uses the notation AB for intersection, A' for complement, A-B for difference, and A∪B for union.

$A - (B - C)$
-> $A(BC')'$ by expanding definition of set difference $X - Y$ as XY'
-> $A(B' \cup C'')$ by DeMorgan's Law $(XY)' = X' \cup Y'$
-> $A(B' \cup C)$ by simplifying double complement $X'' = X$
-> $AB' \cup AC$ by distributive law $X(Y \cup Z) = XY \cup XZ$
-> $(A - B) \cup AC$ by recognizing XY' as definition of $X - Y$

RULE218: $(= S\ (\text{filter } S\ P)) \rightarrow (\Rightarrow (\text{in } x\ S)\ P_x)$, where x is implicitly universally quantified

```
(= (CARDS-IN-HAND P0) (FILTER (CARDS-IN-HAND P0) OUT))
9:28 --- [by RULE218] --->
(=> (IN X2 (CARDS-IN-HAND P0)) (OUT X2))
```

5.8.5. Translation: Summary

Translation moves a problem through a space of equivalent expressions. Its purpose is to move the problem into a region where known methods for operationalization and analysis can be applied. Each kind of translation corresponds to a different direction in this space:

1. *Elaboration* moves *downward* to a lower level of detail, often by expanding definitions.
2. *Recognition* moves *upward* from primitive concepts to more domain-specific ones.
3. *Lateral rephrasing* moves *sideways* at more or less the same level of detail.

Thus translation is a *change of problem representation* based on change of vocabulary. In practice, translation rules are used together with other rules to map problems to available methods.

5.9. Problem Reduction

A common problem-solving strategy is

Reduce the problem to an easier one.

Problem reduction differs from simplification by its heuristic nature. A *simplified* problem is always semantically equivalent to the original one and at least as easy to solve; thus simplification is both *logically valid* and *heuristically useful.* The rules in this section do not enjoy these properties: they may not be guaranteed to preserve equivalence, or to be useful even if applicable.

The term "problem reduction" is often used in a sense that encompasses the idea of problem separation discussed earlier (§5.7). In contrast, the term is used here to refer to the transformation of a problem into *one* eaiser problem, rather than into a collection of subproblems. Thus a problem reduction rule has the general form e -> e' if C, where e' is easier than e in some sense. Such rules can be classified according to the difficulty of testing the condition C.

One category contains rules where the condition C is tested by analysis, *i.e.*, by recursively invoking the whole collection of analysis rules to evaluate C. These rules, discussed in (§5.9.1), check for special cases, such as necessary (§5.9.1.2) or sufficient (§5.9.1.1) conditions on e.

Another category, treated in (§5.9.2), contains reduction rules where C can be tested without analysis. Some of these rules preserve semantic equivalence, but unlike simplification rules, should not always be applied (§5.9.2.1). Others violate semantic invariance by producing a sufficient (§5.9.2.2) or necessary (§5.9.2.3) condition e' on the original expression e, or some other non-equivalent expression (§5.9.2.4). (§5.9.2.2.1) discusses reduction to a sufficient condition in terms of *relevant implication.*

The above distinctions between different classes of reduction rules are blurred in practice (§5.9.3). For example, a rule condition can be treated as an untested assumption (§5.9.3.1) or reduced to a simpler unproved condition (§5.9.3.2), which can be treated as a subgoal or as a restriction on the applicability of the eventual solution. Relationships between different kinds of reduction rules are summarized in (§5.9.4).

5.9.1. Reduction Based on Analysis: Checking for a Special Case

Some reduction rules are of the form e -> e' if C, where the transformation e -> e' is *valid* (preserves semantic equivalence) in the *special case* described by C, and evaluating C requires analysis. An example of a reduction rule that checks for a special case is

RULE362: (Q x S [... (in x S') ...]) -> (Q x S [...]) if (subset S S')

RULE362 is used in DERIV6 to simplify the expression

```
(EXISTS C3 (CARDS-PLAYED) (AND [IN C3 (CARDS)] [HAS-POINTS C3]))
```

Verifying the rule condition requires analysis:

```
(SUBSET (CARDS-PLAYED) (CARDS))
6:18 --- [FACT by RULE363] --->
T
```

Reducing the expression allows it to be simplified and recognized as a familiar concept:

```
(EXISTS C3 (CARDS-PLAYED) (AND [HAS-POINTS C3]))
6:20                         --- [SIMPLIFY by RULE178] --->
(EXISTS C3 (CARDS-PLAYED) (HAS-POINTS C3))
6:21 --- [RECOGNIZE by RULE123] --->
(HAVE-POINTS (CARDS-PLAYED))
6:22 --- [RECOGNIZE by RULE123] --->
(TRICK-HAS-POINTS)
```

Another special case reduction rule is

RULE19: (diff [join S_1 ... S ... S_n] S) -> (join S_1 ... S_n) if (disjoint S_1 ... S ... S_n)

RULE19 is used in DERIV4 to make the reduction

```
(DIFF [UNION (HANDS (PLAYERS))
             (PILES (PLAYERS))
             (SET DECK POT HOLE)]
      [HANDS (PLAYERS)])
4:16-18 --- [REDUCE by RULE19, analysis] --->
(UNION (PILES (PLAYERS)) (SET DECK POT HOLE))
```

RULE19's condition is verified by analysis to check the validity of applying it to the expression:

```
(DISJOINT (HANDS (PLAYERS))
          (PILES (PLAYERS))
          (SET DECK POT HOLE))
4:17 --- [EVAL by RULE346] --->
T
```

Other special case reduction rules are listed below with examples of their use:

RULE300: (next (f S)) -> (f (change S)) if (R [f (change S)] [f S]), assuming (next S) includes S, where R is an ordering, (f S) is the R-extremum of S, and (change S) = (diff (next S) S)

```
(NEXT (HIGHEST CANTUS-1))
14:55-57 --- [REDUCE by RULE300, analysis] --->
(HIGHEST (CHANGE CANTUS-1))
since (HIGHER (HIGHEST (CHANGE CANTUS-1)) (HIGHEST CANTUS-1))
assuming (SUBSET CANTUS-1 (NEXT CANTUS-1))
```

RULE324: (= [f S_1] [f S_2]) -> (in [f S_1] S_2) if (subset S_2 S_1), where (f S) is an extremum of S

```
(= [HIGHEST-IN-SUIT-LED (PROJECT CARD-OF (PLAYERS))]
   [HIGHEST-IN-SUIT-LED CARDS-PLAYED1])
2:63-64 --- [by RULE324, assumption] --->
(IN [HIGHEST-IN-SUIT-LED (PROJECT CARD-OF (PLAYERS))]
    CARDS-PLAYED1]))
assuming (SUBSET CARDS-PLAYED1 (PROJECT CARD-OF (PLAYERS)))
```

RULE325: (= [S_1 i] [S_2 i]) -> (in i (indices-of S_2)) if (subset S_2 S_1), where (S i) = (nth S i)

```
(= [(PROJECT CARD-OF (PLAYERS)) ME] [CARDS-PLAYED1 ME])
2:66-67 --- [by RULE325, ASSUME by RULE32] --->
(IN ME (INDICES-OF CARDS-PLAYED1))
assuming (SUBSET CARDS-PLAYED1 (PROJECT CARD-OF (PLAYERS)))
```

5.9.1.1 Checking a Sufficient Condition

An important special case of checking for a special case is a rule that checks for a *sufficient condition*, *i.e.*, has the form c -> T if C, where C is a sufficient condition for c. One such rule is

RULE336: (=> [P S_1] [P S_2]) -> T if (subset S_1 S_2), where P = (lambda (S) (exists x S R_x))

RULE336 is used in DERIV2 to evaluate

```
(=> [HAVE-POINTS (PREFIX CARDS-PLAYED1 ANY)]
    [HAVE-POINTS CARDS-PLAYED1])
2:97-99 --- [REDUCE by RULE336, analysis] --->
T
```

Here ANY is lazy notation for a universally quantified (non-negative integer) variable. The analysis in this example consists of verifying that the special case holds:

```
(SUBSET (PREFIX CARDS-PLAYED1 ANY) CARDS-PLAYED1)
2:98 --- [EVAL by RULE337] --->
T
```

Another rule that checks for a sufficient condition is

RULE383: (=> [R e_1 e] [R e_2 e]) -> T *if* (R e_2 e_1), where R is transitive

The validity of RULE383 follows from the definition of transitivity:

R *is transitive if* (R e_2 e_1) *and* (R e_1 e) *implies* (R e_2 e)

Whenever the rule condition (R e_2 e_1) is true, (R e_1 e) implies (R e_2 e). Thus the rule condition (R e_2 e_1) is a sufficient condition for (=> [R e_1 e] [R e_2 e]). However, it is *not* a necessary condition. For example, if (R x e) is false for all x, then (=> [R e_1 e] [R e_2 e]) is vacuously true (since false implies anything), even if the condition (R e_2 e_1) is false. (Consider, say, e = Ace, e_1 = King, e_2 = Queen, and R = HIGHER.) RULE383 is used twice in DERIV13:

```
(=> [=< (MELODIC-RANGE (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))))
        (MAJOR 10)]
    [=< (MELODIC-RANGE PROJECT1) (MAJOR 10)])
13:45-48 --- [REDUCE by RULE383, analysis] --->
T
since (=< (MELODIC-RANGE PROJECT1)
          (MELODIC-RANGE (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))))

(=> [=< (#OCCURRENCES CLIMAX1
                      (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))))
     1)
    [=< (#OCCURRENCES CLIMAX1 PROJECT1) 1))
13:93-100 --- [REDUCE by RULE383, analysis] --->
T
since (=< (#OCCURRENCES CLIMAX1 PROJECT1)
          (#OCCURRENCES CLIMAX1
                        (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))))
```

5.9.1.2 Checking a Necessary Condition

Another special case of checking for special cases is *checking for necessary conditions*. A rule that does this has the form e -> nil if ~C, where C is a necessary condition for e. Some intersection search rules have this form (§5.6), with the existence of a path through some network as the necessary condition for e. Another rule of this form checks a necessary condition for an element to be in a set, namely that it not lie beyond the extremum defined by an ordering (*e.g.*, if the richest member of a club is worth a million dollars, J. Paul Getty doesn't belong to it):

RULE304: (in e S) -> nil if (R e (f S)), where R is an ordering and (f S) is R-extremum of S

RULE304 says a tone can't have been used in a cantus if adding it alters the climax:

```
(IN (NEXT NOTE) CANTUS-1)
14:73-75 --- [REDUCE by RULE304, analysis] --->
NIL
```

Here R = HIGHER, f = HIGHEST. The analysis checks that the added note is higher than the current climax:

```
(HIGHER (NEXT NOTE) (HIGHEST CANTUS-1))
14:74 --- [by a premise of the case under consideration] --->
T
```

5.9.2. Reduction Without Analysis

Some reduction rules have conditions that can be tested without analysis. Of these rules, some preserve semantic equivalence but do not qualify as simplification rules because there are situations in which applying them is a bad idea. Others transform expressions into simpler but non-equivalent ones, typically sufficient conditions.

5.9.2.1 Heuristic Equivalence-Preserving Reduction

Some reduction rules preserve semantic equivalence but are not always useful. The *recognition* rules discussed earlier (§5.8.3) are like this. They transform expressions into simpler equivalent ones, but should not be applied whenever applicable, *e.g.*, immediately after expanding the definition of a concept. Another example of a *heuristic equivalence-preserving reduction* rule is

RULE213: ([lambda (x_1 ... x_n) e] e_1 ... e_n) -> e',
where e' is the result of substituting e_1 ... e_n for x_1 ... x_n throughout e

This rule is useful in DERIV14:

```
([LAMBDA (J) (NOTE (+ J 1))] T1)
14:86 --- [SIMPLIFY by RULE213] --->
(NOTE (+ T1 1))
```

However, applying it whenever possible would prevent the following important step in DERIV2:

```
(HSM1 WITH
   (REFORMULATED-PROBLEM :
      ([LAMBDA (CARDS-PLAYED1)
          (AND [HAVE-POINTS CARDS-PLAYED1]
               [= (CARDS-PLAYED1 ME)
                  (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)]])]
       (CARDS-PLAYED))))
14:22 --- [by RULE317] --->
HSM1 <- SOLUTION-TEST :
          (LAMBDA (CARDS-PLAYED1)
             (AND [HAVE-POINTS CARDS-PLAYED1]
                  [= (CARDS-PLAYED1 ME)
                     (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)]))
```

The reason is that the reduction made by RULE213 would render RULE317 inapplicable:

```
([LAMBDA (CARDS-PLAYED1)
    (AND [HAVE-POINTS CARDS-PLAYED1]
         [= (CARDS-PLAYED1 ME)
            (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)])]
 (CARDS-PLAYED)))
--- [by RULE213] --->
(AND [HAVE-POINTS (CARDS-PLAYED)]
     [= ((CARDS-PLAYED) ME)
        (HIGHEST-IN-SUIT-LED (CARDS-PLAYED))])
```

Thus it would not be safe to incorporate RULE213 in a canonicalization procedure applied automatically after every transformation, since doing so would make it impossible to get expressions into the non-canonical form required by RULE317 (§5.2.5).

Other equivalence-preserving reduction rules are listed below with examples of their use.

RULE215: (lambda (x) (f x)) -> f

```
(LAMBDA (EXISTS1) (NOT EXISTS1))
9:15 --- [SIMPLIFY by RULE215] --->
NOT
```

RULE290: (R [+ e k_1] k_2) -> (R e k), where R is an arithmetic relation and $k = k_1 - k_2$

```
(= [+ (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1] 1)
14:27 --- [REDUCE by RULE290] --->
(= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 0)
```

RULE374: (before A B) -> B, where B is a boolean constant (T or nil)

```
(BEFORE (CURRENT ROUND-IN-PROGRESS) NIL)
10:20 --- [EVAL by RULE374] --->
NIL
```

5.9.2.2 *Reduction to a Sufficient Condition*

Some problem reduction rules e -> e' don't guarantee that e' is semantically equivalent to e. They typically guarantee only that e' is a *sufficient condition* for e, *i.e.*, that e' implies e. Such transformations are common in *planning, i.e.*, transforming a goal into a plan whose execution is *sufficient* to achieve it (the generated plan need not be the *only* way to achieve the goal). *Partial matching* often reduces an expression to a sufficient condition (§5.3.3).

Reduction to a sufficient condition is illustrated in DERIV11, where the problem is to find a sufficient condition for deducing that an opponent `P0` is void in suit `S0`. The chosen approach takes a known fact about the game, namely `(LEGAL P1 (CARD-OF P1))`, and determines when it implies `(VOID P0 S0)` (§2.6.2). The goal is to find an *evaluable sufficient condition* for

```
        (=> [LEGAL P1 (CARD-OF P1)] [VOID P0 S0])
```

The first step is to expand the definition of LEGAL:

```
                (LEGAL P1 (CARD-OF P1))
11:7            --- [ELABORATE by RULE124] --->
                (AND [HAS P1 (CARD-OF P1)]
                     [=> (LEADING P1)
                         (OR [CAN-LEAD-HEARTS P1]
                             [NEQ (SUIT-OF (CARD-OF P1)) H])]
                     [=> (FOLLOWING P1)
                         (OR [VOID P1 (SUIT-LED)]
                             [IN-SUIT (CARD-OF P1) (SUIT-LED)])])
```

The next three steps *reduce* the expression:

```
        (=> [AND [HAS P1 (CARD-OF P1)]
                 [=> (LEADING P1)
                     (OR [CAN-LEAD-HEARTS P1]
                         [NEQ (SUIT-OF (CARD-OF P1)) H])]
                 [=> (FOLLOWING P1)
                     (OR [VOID P1 (SUIT-LED)]
                         [IN-SUIT (CARD-OF P1) (SUIT-LED)])]]
            [VOID P0 S0])

11:8    --- [REDUCE by RULE224] --->
        (=> [=> (FOLLOWING P1)
                (OR [VOID P1 (SUIT-LED)] [IN-SUIT (CARD-OF P1)
                                                  (SUIT-LED)])]
            [VOID P0 S0])

11:9    --- [REDUCE by RULE226] --->
        (AND [FOLLOWING P1]
             [=> [OR [VOID P1 (SUIT-LED)] [IN-SUIT (CARD-OF P1)
                                                    (SUIT-LED)]]
                 [VOID P0 S0]])

             (=> [OR [VOID P1 (SUIT-LED)] [IN-SUIT (CARD-OF P1)
                                                    (SUIT-LED)]]
                 [VOID P0 S0])
11:10        --- [REDUCE by RULE225] --->
             (AND [NOT (OR [IN-SUIT (CARD-OF P1) (SUIT-LED)])]
                  [=> [VOID P1 (SUIT-LED)] [VOID P0 S0]])
```

At this point, partial matching can reduce the implication to a conjunction of equalities:

```
                     (=> [VOID P1 (SUIT-LED)] [VOID P0 S0])
11:11                --- [DISTRIBUTE by RULE43] --->
                     (AND [= P1 P0] [= (SUIT-LED) S0])
```

Solving for P1 and simplifying gives the desired expression:

```
        (AND [FOLLOWING P1]
             [AND [NOT (OR [IN-SUIT (CARD-OF P1) (SUIT-LED)])]
                  [AND [= P1 P0] [= (SUIT-LED) S0]]])
  11:12-17 --- [by analysis] --->
        (AND [NOT (IN-SUIT (CARD-OF P0) (SUIT-LED))]
             [= (SUIT-LED) S0]
             [FOLLOWING P0])
```

In short, a sufficient condition for deducing player P0 to be void in suit S0 is that P0 play a card in a suit other than the suit led when the suit led is S0 and P0 is not leading. (The last condition follows from the first, of course: by definition, the leader's card is in the suit led).

5.9.2.2.1 Relevant Implication and Reduction to a Sufficient Condition

Reduction played a key role in the above example by transforming the expression so as to enable partial matching. Reduction doesn't always preserve logical equivalence; often it transforms e -> e' where e' only *implies* e. The non-equivalence shows up when e = T but e' = NIL, which occurs in the following rules for the cases indicated:

RULE224: (=> [and C_1 ... C ... C_n] [P ...]) -> (=> C [P ...]) if P occurs in C
-- take C_1 = (P ...) = NIL

RULE225: (=> [or ... C ...] [P ...]) -> (and [not (or ...)] [=> C (P ...)]) if P occurs in C
-- take (P ...) = C' = T for some C' other than C in (or ... C ...)

RULE226: (=> [=> R C] [P ...]) -> (and R [=> C (P ...)]) if P occurs in C
-- take R = NIL, (P ...) = T

The non-equivalence has to do with the *relevance* of P to C. For instance, if (and C_1 ... C ... C_n) implies (P ...) in RULE224, it should be *because C is true*, since P occurs in C and is therefore presumably relevant to it. Similarly, if (or ... C ...) implies (P ...) in RULE225, it should be because C is true, which must certainly be the case if the other terms in the disjunction are false. Although some of FOO's rules treat the connective => as standard predicate calculus implication (where, for instance, NIL => P is true for any P), the reduction rules above treat it more like the *relevant implication* of *relevance logic* [Anderson 75], where P => Q is false unless P is *relevant* to Q.

Other rules for reduction to a sufficient condition are listed below with examples of their use.

RULE172: (in (f e) (project f S)) -> (in e S) -- *equivalent iff f is injective*

```
(IN (HAND ME) (PROJECT HAND (PLAYERS)))
4:23 --- [REMOVE-QUANT by RULE172] --->
(IN ME (PLAYERS))
```

```
(IN (CARD-OF ME) (PROJECT CARD-OF (PLAYERS)))
5:23 --- [REMOVE-QUANT by RULE172] --->
(IN ME (PLAYERS))
```

RULE193: (= P_x (in x S)) -> (= S [set-of x [domain x P_x] P_x])

```
(= (IN-SUIT C1 S0) (IN C1 S1))
9:7 --- [by RULE193] --->
(= S1 [SET-OF C1 [DOMAIN C1 (IN-SUIT C1 S0)] (IN-SUIT C1 S0)])
```

RULE193 initiates a search for the domain (*i.e.*, *type*) (§5.5.3) of variable x in order to find an evaluable superset of S (§2.3.1.4).

RULE277: (change (f S)) -> (R [f (change S)] [f S]) -- *equivalent if S is expanding*

```
(CHANGE (HIGHEST CANTUS-1))
14:42 --- [REDUCE by RULE277] --->
(HIGHER [HIGHEST (CHANGE CANTUS-1)] [HIGHEST CANTUS-1])
```

Here (f S) is the extreme element of S with respect to ordering R, and CHANGE is ambiguous: (change (f S)) is true iff (f S) ≠ (next (f S)), but (change S) denotes (diff (next S) S).

5.9.2.3 Reduction to a Necessary Condition

A reduction rule e -> e' may guarantee only that e' is a *necessary condition* for e. Such is

RULE369: (in e (set-of x S P_x)) -> P_e -- *equivalent if (in e S)*

RULE369 is used in DERIV9 to make the reduction

```
(IN X2 (SET-OF C4 (CARDS) (HAS P0 C4)))
9:30 --- [REMOVE-QUANT by RULE369] --->
(HAS P0 X2)
```

The reduced expression does not test whether X2 is in (CARDS). RULE369 collapses into a single step the sequence

(in e (set-of x S P_x))
-> (and [in e S] P_e) -- *propagate S-P split (preserves equivalence)*
-> (and T P_e) -- *assume (in e S)*
-> P_e -- *simplify*

Another rule for reducing an expression to a necessary condition is

RULE350: (achieve (and ... P ...)) -> (achieve (and ...))

RULE350 preserves equivalence if the right-hand conjunction implies P.

It is used in DERIV3 to eliminate an unnecessarily specific clause generated in planning:

```
(ACHIEVE (AND [PLAY (QSO) C3]
              [=> (FOLLOWING (QSO)) (IN-SUIT C3 S)]
              [IN C3 (DIFF (CARDS) (SET QS))]))
3:81 --- [by RULE350] --->
(ACHIEVE (AND [PLAY (QSO) C3]
              [=> (FOLLOWING (QSO)) (IN-SUIT C3 S)]))
```

The condition (IN C3 (DIFF (CARDS) (SET QS))) can be dropped since the goal of flushing the Queen does not require C3 $\neq$ QS.

Another rule reduces an expression to a necessary condition by substituting e_2 for e_1 based on an equality $e_1 = e_2$ given in the problem:

RULE354: (and ... [= e_1 e_2] ... [P ... e_1 ...] ...) -> (and ... [= e_1 e_2] ... [P ... e_2 ...] ...)

RULE354 substitutes ME for (LEADER) in DERIV3 and DERIV5:

```
(AND [= (LEADER) ME] [SPADE! (CARD-OF (LEADER))])
3:95 --- [by RULE354] --->
(AND [= (LEADER) ME] [SPADE! (CARD-OF ME)])

(AND [= (LEADER) ME]
     [SPADE! (CARD-OF ME)]
     [NOT (AND [IN QS (CARDS-PLAYED)]
               [AND [= (SUIT-OF (CARD-OF ME))
                       (SUIT-OF (CARD-OF (LEADER)))]
                    [FORALL X1
                            (CARDS-IN-SUIT-LED (CARDS-PLAYED))
                            (NOT (HIGHER X1 (CARD-OF ME)))]])])
5:32 --- [by RULE354] --->
(AND [= (LEADER) ME]
     [SPADE! (CARD-OF ME)]
     [NOT (AND [IN QS (CARDS-PLAYED)]
               [AND [= (SUIT-OF (CARD-OF ME))
                       (SUIT-OF (CARD-OF ME))]
                    [FORALL X1
                            (CARDS-IN-SUIT-LED (CARDS-PLAYED))
                            (NOT (HIGHER X1 (CARD-OF ME)))]])])
```

RULE354 can be characterized as *propagating an assumption* (§5.4.3).

5.9.2.4 Reduction to a Non-equivalent Expression

For some reduction rules e -> e', e' is neither necessary nor sufficient for e, as in

RULE238: (choose c S E_c) -> E_c

RULE238 is used to discard superfluous information preventing recognition of PROJECT:

```
(EACH I2 (LB:UB 1 (CANTUS-LENGTH))
   [CHOOSE (NOTE I2) (TONES) (NOTE I2)])
13:13 --- [REDUCE by RULE238] --->
(EACH I2 (LB:UB 1 (CANTUS-LENGTH)) (NOTE I2))
13:14 --- [RECOGNIZE by RULE123] --->
(PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))
```

The c in (choose c S E_c) is by convention a composite name containing enough information to retrieve the choice event itself (§2.7); thus RULE238 doesn't really lose information in the way that a transformation like (exists x S P_x) -> P_x would.

Another reduction transformation that doesn't preserve semantic equivalence is given by

RULE366: (achieve (... (previous e) ...)) -> (achieve (... e ...))

RULE366 is used in DERIV7 in constructing a plan to get the lead:

```
(ACHIEVE (LEADING ME))
7:1-2 --- [ELABORATE by RULE124] --->
(ACHIEVE (= (PREVIOUS (TRICK-WINNER)) ME))
7:3 --- [by RULE366] --->
(ACHIEVE (= (TRICK-WINNER) ME))
```

RULE366 expresses the simple idea that

What is now the present will become the past in the future.

You can't make something happen yesterday (without travelling back through time), but if you make it happen today, tomorrow you will be able to say "it was true yesterday." In the example at hand, you can't make yourself win the previous trick, but if you win the current trick, you'll lead the next one. The explanation is more complicated than the idea; English is an awkward language for expressing concepts like "what 'yesterday' will mean tomorrow."

5.9.3. Discussion

This section has thus far classified reduction rules e -> e' if C according to two criteria:

1. The difficulty of evaluating C: does it require analysis?
2. The logical relationship of e' to e: are they equivalent? does one imply the other?

In some ways these distinctions are superficial. In particular, one can change the classification of a rule e -> e' if C simply by making the condition C an assumption or incorporating C in e'. The following discussion explains how some of these connections are exploited in FOO.

5.9.3.1 Reduction Based on Assumption

Consider an equivalence-preserving rule of the form

e -> e' if C

Assume the rule requires analysis to check for the special case C (§5.9.1). Now consider the rule

e -> e' *assuming* C

Although almost identical to the first rule, it is classified quite differently, since it requires no analysis and is not guaranteed to preserve semantic equivalence (§5.9.2.3).

FOO has a rule that allows it to treat the first kind of rule as if it were the second:

RULE32: e -> e' assuming C', if the condition C of some rule e -> e' has been reduced to C'

RULE32 is used in DERIV2 to make the reduction

```
(= (HIGHEST-IN-SUIT-LED (PROJECT CARD-OF (PLAYERS)))
   (HIGHEST-IN-SUIT-LED CARDS-PLAYED1))
2:63-64 --- [REDUCE by RULE322, ASSUME by RULE32] --->
(IN (HIGHEST-IN-SUIT-LED (PROJECT CARD-OF (PLAYERS))
    CARDS-PLAYED1)
```

This is done by assuming the unproved condition of RULE322:

```
(SUBSET CARDS-PLAYED1 (PROJECT CARD-OF (PLAYERS)))
2:64 --- [ASSUME by RULE32] --->
T
```

RULE32 is also used in DERIV9 to make the refinement

```
(PR-DISJOINT (CARDS-IN-HAND P0) (CARDS-IN-SUIT S0) (CARDS))
9:27-37 --- [REFINE by RULE200, REDUCE by analysis] --->
(PR-DISJOINT (CARDS-IN-HAND P0) (CARDS-OUT-IN-SUIT S0) (CARDS-OUT))
```

Here the reduced condition of RULE200 is assumed:

```
(= (CARDS-IN-HAND P0) (FILTER (CARDS-IN-HAND P0) OUT))
9:28-36 --- [REDUCE by analysis] --->
(IN P0 (OPPONENTS ME))
9:37 --- [ASSUME by RULE32] --->
T
```

RULE32 could be used as part of the following strategy for making *plausible assumptions:*

1. Given a potentially useful reduction e -> e' if C, try to *prove* C.

2. If you reduce C to C' but can't prove or disprove C', *assume* C' and reduce e to e'.

The idea here is that an incomplete proof of C shouldn't be wasted, since completing it may only require a fact not in the knowledge base, like `(SUBSET CARDS-PLAYED1 (CARDS-PLAYED))` in the first example, or a reasonable restriction on the solution, like `(IN P0 (OPPONENTS ME))` in the second. One way to exploit such as incomplete proof is to assume C' and reduce e to e'. An alternative is to use some other means to ascertain C'. For example, an empirical approach would test C' in several actual task situations to see if it were true in all of them (§8.1.7).

5.9.3.2 Exploiting Special Case Near-Misses

Often a special-case rule e -> e' if C is *almost* applicable to a given expression, *i.e.*, C reduces to a much simpler condition C'. Even when C can't be proved, the rule can still be useful in such *special-case near-miss* situations if C' is treated as an *assumption*, as a *restriction* on the solution, or as an *additional subgoal* to be achieved.

5.9.3.2.1 Reducing a Goal to a Feasible Subgoal

Consider a rule that requires analysis to check a sufficient condition C for e (§5.9.1.1):

e -> T if C

Consider also a rule that reduces e to a sufficient condition without analysis and is not guaranteed to preserve semantic equivalence (§5.9.2.2):

e -> C

The second rule *subsumes* the first in the sense that if the first rule can reduce e to T, so can the second. In the first case C would reduced to T during verification of the rule condition, while in the second this would occur after the rule was applied. The analysis required would be identical in the two cases, the only difference being whether it was carried out as a subgoal or at the same level as the original problem.

FOO has a rule that allows it to treat the first kind of rule as if it were the second:

RULE151: P -> R, where R is the result of reducing an unproved sufficient condition for P

RULE151 is used in DERIV6 to make the reduction

```
(HIGHER (FIND-ELT (CARDS-IN-SUIT-LED (CARDS-PLAYED)))
        (CARD-OF ME))
6:44 --- [TRY-THIS by RULE142] --->
(AND [IN CARD1 (CARDS-IN-SUIT-LED (CARDS-PLAYED))]
     [HIGHER CARD1 (CARD-OF ME)])
6:45-61 --- [reduce by analysis, RULE151] --->
(HIGHER DK (CARD-OF ME))
```

The goal of making (CARD-OF ME) lower than some card played in the suit led is set up as an equation on the variable CARD1. Substituting DK for CARD1 based on the previous assumption that the King of diamonds is led (§5.4.4) reduces the equation to (HIGHER DK (CARD-OF ME)). This unprovable condition is adopted in place of the original goal, and the advice "avoid taking points" is operationalized as "play a card lower than the King."

This example illustrates the following strategy for generating *feasible subgoals*:

1. Given a goal G and a sufficient condition rule G -> T if C, try to *prove* C.
2. If you reduce C to C' but can't prove or disprove C', *reduce* G to C' and *achieve* C'.

The idea here is that an unsuccessful attempt to *prove* a desired condition may reduce it to a condition that satisfies the same original goal but is easier to *achieve*.

5.9.3.2.2 Finding Restricted Solutions

RULE151 also helps find *restricted solutions* to problems. For example, a problem in DERIV2 is to find a necessary condition for a player P1 to have a card C2. One such condition is that the card be OUT, but this solution only applies when P1 is an opponent of player ME:

```
(=> [HAS P1 C2] [OUT C7])

2:29             --- [ELABORATE by RULE124] --->
(=> [HAS P1 C2] [EXISTS P4 (OPPONENTS ME) (HAS P4 C7)])

2:30 --- [REDUCE by RULE322] --->
(AND [IN P4 (OPPONENTS ME)] [= P1 P4] [= C2 C7])

2:31-37 --- [reduce by analysis, RULE151] --->
(IN P1 (OPPONENTS ME))
```

Later in DERIV2, the problem arises of verifying the *monotonicity condition* that moving a particular test earlier in a heuristic search will not cause any potential solutions to be discarded (§3.4.1). The test requires that the card played by player ME be the highest card in the suit led in the card sequence CARDS-PLAYED1. The monotonicity condition reduces to the restriction that the sequence include the card played by player ME:

```
(=> [= ((PROJECT CARD-OF (PLAYERS)) ME)
       (HIGHEST-IN-SUIT-LED (PROJECT CARD-OF (PLAYERS)))]
    [= (CARDS-PLAYED1 ME)
       (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)])

2:62 --- [PARTIAL-MATCH (=> [= ...] [= ...]) by RULE91] --->
(AND [= ((PROJECT CARD-OF (PLAYERS)) ME) (CARDS-PLAYED1 ME)]
     [= (HIGHEST-IN-SUIT-LED (PROJECT CARD-OF (PLAYERS)))
        (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)])

2:63-72 --- [reduce by analysis, RULE151] --->
(IN ME (INDICES-OF CARDS-PLAYED1))
```

This restriction is incorporated into the search so that the test is only applied to sequences containing player ME's card.

This example illustrates the following strategy for generating *restricted solutions*:

1. Given a problem P and a conditional reduction rule P -> P' if C, try to *prove* C.
2. If C reduces to an unproved condition C', *reduce* P to P' and adopt C' as a *restriction*.

The *restricted solution* is one that solves the original problem P only when some additional constraint C' is satisfied. Restricted solutions can be useful when no general solutions can be found. For example, there is no general way to tell whether an opponent is void in a suit, but there are several methods that work in particular special cases, such as when the opponent has previously broken the suit in question (§2.6.2).

5.9.3.2.3 Treating a Reduced Rule Condition as an Additional Subgoal

If the original problem is a goal, it may be possible to treat C' as an *additional subgoal* rather than as a restriction on the applicability of the solution:

1. Given a goal G and a conditional reduction rule G -> G' if C, try to *prove* C.
2. If C reduces to an unproved condition C', *reduce* G to (and G' C') and *achieve* this.

The particular case of G' = T corresponds to the *feasible subgoal* strategy (§5.9.3.2.1).

5.9.4. Problem Reduction: Summary

Problem reduction can profitably be compared to *simplification*. A simplification rule e -> e' if C:

1. Produces an expression e' *simpler* than e, *i.e.*, smaller and easier to solve.
2. Preserves the semantic *equivalence* of e and e'.

3. Requires *no analysis* to evaluate the condition C.
4. *Never hurts* to apply, since it doesn't preclude application of other methods.

In contrast, a problem reduction rule e -> e' if C:

1. Produces an expression e' *easier to analyze* than e but not always smaller.
2. May only guarantee e' to be a *necessary* or *sufficient* condition for e.
3. May *require analysis* to check for the special case C.
4. Is *harmful* when it gives a useless result or prevents application of other methods.

These criteria can be used to classify problem reduction rules:

1. *Analysis-based* reduction rules *check for a special case* and preserve equivalence when it holds.
    a. Rules of the form e -> T if C *check a sufficient condition* C for e.
    b. Rules of the form e -> nil if ~C *check a necessary condition* C for e.
2. *Analysis-free* reduction rules can be harmful to apply, or fail to preserve equivalence.
    a. *Heuristic equivalence-preserving reduction* interferes with other rules if used indiscriminately.
    b. *Reduction to a sufficient condition* is useful in planning and relates to *relevant implication.*
    c. *Reduction to a necessary condition* exploits implicit or explicit assumptions.
    d. *Reduction to a non-equivalent expression* transforms the problem to make it solvable.

Problem reduction subsumes some kinds of rules described by other names. For example, *recognition* (§5.8.3) is *heuristic equivalence-preserving reduction*, since it interferes with other methods, *e.g.*, elaboration. *Partial matching* (§5.3) is often *reduction to a sufficient condition*. *Intersection search* (§5.6) is often used to *check a necessary condition.*

The criteria used in this classification are somewhat superficial in that a rule classified in one category may be functionally very similar to one in a completely different category. FOO has some rules whose effect is the same as transforming a rule e -> e' if C from one category to another:

1. *Reduction by assumption* eliminates analysis by assuming C, possibly violating equivalence.
2. *Reduction by special-case near-miss* reduces C to an *assumption*, *restriction*, or *subgoal.*

5.10. Restructuring

A common tactic in solving a problem is to

Restructure the problem so that other methods can be applied.

Restructuring an expression means rearranging its components, as opposed to *translation* (§5.8), which reformulates it in terms of different concepts. Thus a restructured expression usually contains the same symbols as the original, although minor changes in vocabulary may be required to preserve syntactic well-formedness or semantic consistency. For example, two restructuring rules are:

RULE58: (during [each x S E_x] A) -> (exists x S [during E_x A]) -- *extract* EACH

RULE100: (during A [for-some x S E_x]) -> (exists x S [during A E_x]) -- *extract* FOR-SOME

These rules help in analyzing whether player ME takes points when the winner takes the trick:

```
(DURING [TAKE-TRICK (TRICK-WINNER)] [TAKE-POINTS ME])

6:7-8   --- [ELABORATE by RULE124] --->
(DURING [EACH C3 (CARDS-PLAYED) (TAKE (TRICK-WINNER) C3)]
        [FOR-SOME C1 (POINT-CARDS) (TAKE ME C1)])

6:9 --- [RESTRUCTURE by RULE58] --->
(EXISTS C3 (CARDS-PLAYED)
   (DURING [TAKE (TRICK-WINNER) C3]
           [FOR-SOME C1 (POINT-CARDS) (TAKE ME C1)]))

6:10 --- [RESTRUCTURE by RULE100] --->
(EXISTS C3 (CARDS-PLAYED)
   (EXISTS C1 (POINT-CARDS)
      (DURING [TAKE (TRICK-WINNER) C3] [TAKE ME C1])))

6:11 --- [REDUCE by RULE43] --->
(EXISTS C1 (CARDS-PLAYED)
   (EXISTS C1 (POINT-CARDS)
      (AND [= (TRICK-WINNER) ME] [= C3 C1])))
```

The idea here is to reduce the problem by partial matching. First both arguments of the initial expression are elaborated in terms of a common concept TAKE, introducing the quantifiers EACH and FOR-SOME. To put the expression in the desired form (DURING [TAKE ...] [TAKE ...]), the quantifiers are moved outside by the restructuring rules, changing EACH and FOR-SOME to EXISTS in the process. The restructured expression is reduced by partial matching, and the resulting expression is subsequently simplified.

RULE58 and RULE100 perform a kind of restructuring transformation called *function transposition* because they *transpose* the order in which two functions f and g are applied, *i.e.*, they have the form (f ... [g ...] ...) -> (g' ... [f ...] ...)

In steps 6:9-10, f = `DURING`, g = `EACH` and `FOR-SOME` respectively, and g' = `EXISTS`.

This section is organized as follows. (§5.10.1) describes FOO's rules for function transposition. (§5.10.2) describes rules that *transfer* information from e_i to e_j in an expression of the form (f e_1 ... e_n). (§5.10.3) describes rules that convert *functions* into *functionals* and *vice versa* by moving information between an argument e_i and the function f, which may be a non-atomic expression. (§5.10.4) summarizes the differences and similarities among the three kinds of restructuring rules.

5.10.1. Function Transposition

In general, *function transposition* is a restructuring transformation of the form

(f ... [g ...] ...) -> (g' ... [f ...] ...)

If the initial expression is (f e_1 ... e_n), the function symbol f may be transposed with g in one, some, or all the arguments e_i. A sub-expression (g ...) may be a *top-level* component e_i of (f e_1 ... e_n), or it may be *nested* within e_i. Transposition with g = g' is called *pure*, as distinguished from *quasi-transposition.*

Transposition has some special cases. *Distribution* transposes f with g in two or more arguments:

(f ... [g e_1 ... e_n] ...) -> (g' [f ... e_1 ...] ... [f ... e_n ...])

This kind of transformation is useful for propagating splits in problems (§5.7.5). The inverse of distribution is *collection*:

(f [g ... e_1 ...] ... [g ... e_n ...]) -> (g' ... [f e_1 ... e_n] ...)

Transposing f with g in a single sub-expression is called *embedding* f or *extracting* g:

(f e_1 ... [g s_1 ... s_k] ... e_n) -> (g' s_1 ... [f e_1 ... s_j ... e_n] ... s_k)

If e_i in (f e_1 ... e_i ... e_n) is (g s_1 ... s_j ... s_k), embedding f produces (g' s_1 ... s_j' ... s_k), where s_j' is (f e_1 ... s_j ... e_n). That is, f and all its arguments e_1 ... e_n except e_i are *embedded* in the subexpression e_i = (g s_1 ... s_j ... s_k) by replacing s_j with (f e_1 ... s_j ... e_n). Equivalently, g and all its arguments s_1 ... s_k except s_j are *extracted* from (f e_1 ... e_n) by replacing e_i with s_j, and become top-level components of the new expression, with g possibly changed to g'. Transposition is even harder to describe in English when (g s_1 ... s_k) is *nested* within e_i; the general idea is that it[9]

משפיל גאים ומגביה שפלים מוציא אסירים ופודה ענוים

Fortunately, these transformations can be diagrammed quite simply as splicing operations on trees with labelled nodes, where an expression (f e_1 ... e_n) is represented as a tree with root labelled f and offspring representing e_1 ... e_n. In this representation, function transposition corresponds to a structural transformation described in the early literature[10]:

השפלתי עץ גבה הגבהתי עץ שפל

Figure 5-1 illustrates the three kinds of function transposition described above (distribution, collection, and embedding/extraction).

As a problem-solving tactic, transposition brings one concept towards contact with (direct application to) another by removing an intervening function, either *embedding* it at a deeper level of nesting or *extracting* it to a higher one. In the earlier example, the quantifiers FOR-SOME and EACH were *extracted* to bring DURING into contact with TAKE for partial matching. In the following example from DERIV3, the function DIFF is *embedded* to bring REMOVE-1-FROM in contact with SET-OF so RULE7 can be used:

```
(REMOVE-1-FROM (DIFF [LEGALCARDS (QSO)] [SET QS])
3:39                --- [ELABORATE by RULE124] --->
(REMOVE-1-FROM (DIFF [SET-OF C3 (CARDS) (LEGAL (QSO) C3)]
                     [SET QS]))
3:40          --- [EMBED by RULE5] --->
(REMOVE-1-FROM (SET-OF C3 [DIFF (CARDS) (SET QS)] (LEGAL (QSO) C3)))
3:41 --- [PROPAGATE-SPLIT by RULE7] --->
(UNDO (AND [IN C3 (DIFF (CARDS) (SET QS))] [LEGAL (QSO) C3]))
```

This embedding is accomplished by the transposition rule

RULE5: (diff [set-of x S_1 P_x] S_2) -> (set-of x [diff S_1 S_2] P_x) -- *embed* DIFF

FOO's other transposition rules are listed below with examples of their use. Underlining indicates *correspondence* between rule variables and components of expressions.

[9]bringest low the haughty and raisest up the lowly ... leadest forth the captives and deliverest the meek. [from a Mishnaic (200 B.C.-200 A.D.) prayer in the daily morning service]

[10]I have made low the high tree, have made high the low tree (Ezekiel 17:24).

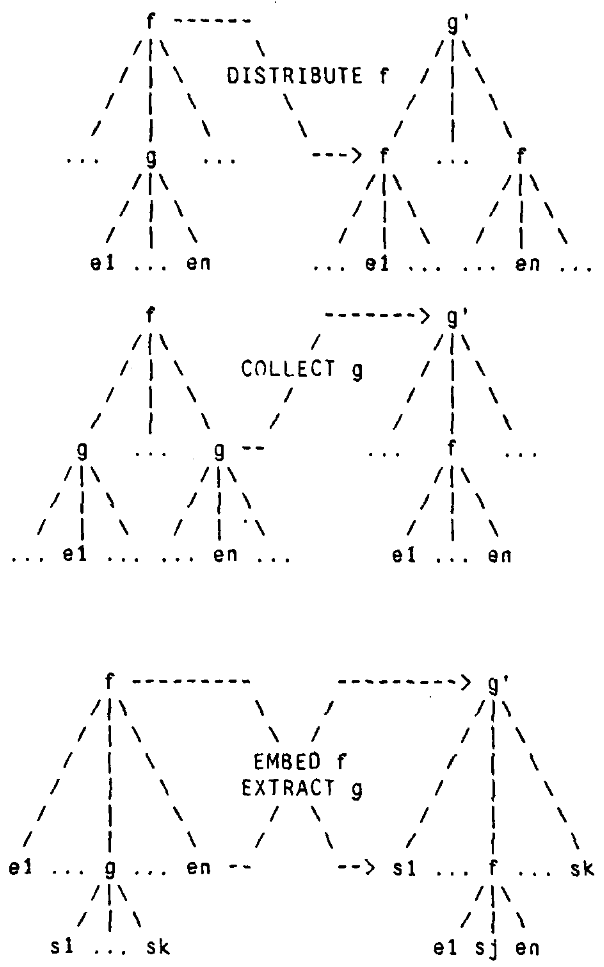

Figure 5-1: Transposition of f and g in (f ... (g ...) ...)

RULE170: (project f [diff S S']) -> (diff [project f S] [project f S']), where f is injective -- *distribute* PROJECT

```
  (PROJECT HAND [DIFF (PLAYERS) (SET ME)])
  4:12 --- [DISTRIBUTE by RULE170] --->
  (DIFF [PROJECT HAND (PLAYERS)] [PROJECT HAND (SET ME)])
```

RULE220: (=> C [Q x S P_x]) -> (Q x S [=> C P_x]) -- *extract Q*

```
(=> [HAS P1 C2]
    [EXISTS C11 (CARDS)
       (AND [HAS P5 C11] [IN-SUIT C11 (SUIT-LED)])])
2:46 --- [REDUCE by RULE220] --->
(EXISTS C11 (CARDS)
   [=> (HAS P1 C2) (AND [HAS P5 C11] [IN-SUIT C11 (SUIT-LED)])])
```

RULE283: (next [g ... e ...]) -> (g ... [next e] ...), where e is a fluent -- *embed* NEXT

```
(NEXT [= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1])
14:6 --- [REDUCE by RULE283] --->
(= [NEXT (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1)] 1)

(NEXT (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1))
14:7 --- [REDUCE by RULE283] --->
(#OCCURRENCES (NEXT (CLIMAX CANTUS-1)) (NEXT CANTUS-1))
```

RULE296: (and ... [not C] ... [not C'] ...) -> (and ... [not (or C C')] ...) -- *collect* NOT

RULE296 generalizes DeMorgan's law (~A and ~B) = ~(A or B).

```
(AND [= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1]
     [NOT (= (NEXT NOTE) (CLIMAX CANTUS-1))]
     [NOT (HIGHER (NEXT NOTE) (CLIMAX CANTUS-1))])
14:47 --- [COLLECT by RULE296] --->
(AND [= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1]
     [NOT (OR [HIGHER (NEXT NOTE) (CLIMAX CANTUS-1)]
              [= (NEXT NOTE) (CLIMAX CANTUS-1)])])
```

RULE329: (P ... [Q x S R_x] ...) -> (Q x S [P ... R_x ...]) -- *extract Q*

```
(=> (IN ME (INDICES-OF CARDS-PLAYED1))
    (AND [IN (CARDS-PLAYED1 ME) (CARDS-IN-SUIT-LED CARDS-PLAYED1)]
         [FORALL P1 (INDICES-OF CARDS-PLAYED1)
            (=> (= (SUIT-OF (CARDS-PLAYED1 P1)) (SUIT-LED))
                (NOT (HIGHER (CARDS-PLAYED1 P1)
                             (CARDS-PLAYED1 ME))))]))
2:81 --- [by RULE329] --->
(FORALL P1 (INDICES-OF CARDS-PLAYED1)
   [=> (IN ME (INDICES-OF CARDS-PLAYED1))
       (AND [IN (CARDS-PLAYED1 ME)
                (CARDS-IN-SUIT-LED CARDS-PLAYED1)]
            [=> (= (SUIT-OF (CARDS-PLAYED1 P1)) (SUIT-LED))
                (NOT (HIGHER (CARDS-PLAYED1 P1)
                             (CARDS-PLAYED1 ME)))])])

(=> (HAS P0 X2) [EXISTS P1 (OPPONENTS ME) (HAS P1 X2)])
9:32 --- [by RULE329] --->
(EXISTS P1 (OPPONENTS ME) [=> (HAS P0 X2) (HAS P1 X2)])

(CAUSE [EXISTS P1 (OPPONENTS ME) (HAS P1 X1)])
10:6 --- [by RULE329] --->
(EXISTS P1 (OPPONENTS ME) [CAUSE (HAS P1 X1)])

(UNDO [EXISTS P3 (OPPONENTS ME) (HAS P3 X1)])
10:27 --- [by RULE329] --->
(EXISTS P3 (OPPONENTS ME) [UNDO (HAS P3 X1)])
```

```
(CAUSE [EXISTS C1 (CARDS) (AND [HAS P0 C1] [IN-SUIT C1 S0])])
12:6 --- [by RULE329] --->
(EXISTS C1 (CARDS) [CAUSE (AND [HAS P0 C1] [IN-SUIT C1 S0])])
```

RULE359: (and [until P A] ...) -> (until P (and A ...)) -- *extract* UNTIL

```
(AND [UNTIL (PLAYED! QS) (ACHIEVE (LEAD-SPADE ME))]
     [ACHIEVE (NOT (DURING (TRICK) (TAKE ME QS)))])
5:3 --- [by RULE359] --->
(UNTIL (PLAYED! QS)
       (AND [ACHIEVE (LEAD-SPADE ME)]
            [ACHIEVE (NOT (DURING (TRICK) (TAKE ME QS)))]))
```

RULE360: (and [achieve P_1] ... [achieve P_n]) -> (achieve (and P_1 ... P_n)) -- *collect* ACHIEVE

```
(AND [ACHIEVE (LEAD-SPADE ME)]
     [ACHIEVE (NOT (DURING (TRICK) (TAKE ME QS)))])
5:4 --- [COLLECT by RULE360] --->
(ACHIEVE (AND [LEAD-SPADE ME]
              [NOT (DURING (TRICK) (TAKE ME QS))]))
```

5.10.2. Transfer

Another kind of restructuring transformation *transfers* information from one component of an expression to another at the same level, *i.e.*, from e_i to e_j in (f e_1 ... e_n). One such rule is

RULE161: (Q x S $P_{[f\,x]}$) -> (Q y [project f S] P_y)

It introduces a *change of variable* from player P1 to location Y1 by *transferring* f from P to S:

```
(OUT QS)
4:2-4 --- [ELABORATE by RULE124] --->
(EXISTS P1 (OPPONENTS ME) (= (LOC QS) [HAND P1]))
4:5 --- [TRANSFER by RULE161] --->
(EXISTS Y1 [PROJECT HAND (OPPONENTS ME)] (= (LOC QS) Y1))
4:6 --- [RECOGNIZE by RULE162] --->
(IN (LOC QS) [PROJECT HAND (OPPONENTS ME)])
4:7 --- [PIGEON-HOLE by RULE169] --->
(NOT
   (IN (LOC QS) [DIFF (RANGE LOC) (PROJECT HAND (OPPONENTS ME))]))
```

The requantified equality is recognized as set membership, allowing the pigeon-hole principle (§2.2) to be applied. Note that the transfer introduces PROJECT as a syntactic side effect.

FOO's other transfer rules are listed below with examples of their use.

RULE13: (Q x [M y S e_y] P_x) -> (Q y S P_{e_y}), where M is injective -- *change of variable*

```
(FORALL INTERVAL1 [DISTRIBUTE I4 (LB:UB 2 (# PROJECT1)
                          (INTERVAL (PROJECT1 (- I4 1)) (PROJECT1 I4))]
   (USABLE-INTERVAL! INTERVAL1))
13:30 --- [REMOVE-QUANT by RULE13] --->
(FORALL I4 (LB:UB 2 (# PROJECT1))
   (USABLE-INTERVAL! (INTERVAL (PROJECT1 (- I4 1)) (PROJECT1 I4))))
```

RULE187: (union ... [set e_1 ... e_k] ... [set e_{k+1} ... e_n] ...) -> (union [set e_1 ... e_n] ...)

```
(UNION [PILES (PLAYERS)] [SET DECK POT HOLE] [SET (HAND ME)])
4:29 --- [SIMPLIFY by RULE187] --->
(UNION [SET DECK POT HOLE (HAND ME)] [PILES (PLAYERS)])
```

RULE219: (forall x [set-of y S P_y] R_x) -> (forall x S [=> P_x R_x])

```
(FORALL X1 [SET-OF C19 CARDS-PLAYED1 (= (SUIT-OF C19) (SUIT-LED))]
   (NOT (HIGHER X1 (CARDS-PLAYED1 ME))))
2:79 --- [by RULE219] --->
(FORALL X1 CARDS-PLAYED1
   [=> (= (SUIT-OF X1) (SUIT-LED))
       (NOT (HIGHER X1 (CARDS-PLAYED1 ME)))])
```

RULE326: (in [S i] S) -> (in i [indices-of S])

```
(IN (CARDS-PLAYED1 ME) CARDS-PLAYED1)
2:69 --- [REDUCE by RULE326] --->
(IN ME (INDICES-OF CARDS-PLAYED1))
```

RULE328: (Q x S P_x) -> (Q i [indices-of S] $P_{[S\ i]}$) -- *change of variable*

```
(FORALL X1 CARDS-PLAYED1
   (=> (= (SUIT-OF X1) (SUIT-LED))
       (NOT (HIGHER X1 (CARDS-PLAYED1 ME)))))
2:80 --- [by RULE328] --->
(FORALL P1 [INDICES-OF CARDS-PLAYED1]
   (=> (= (SUIT-OF [CARDS-PLAYED1 P1]) (SUIT-LED))
       (NOT (HIGHER [CARDS-PLAYED1 P1] (CARDS-PLAYED1 ME)))))

(FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1)))
13:73 --- [by RULE328] --->
(FORALL I5 [INDICES-OF PROJECT1]
   (NOT (HIGHER [PROJECT1 I5] CLIMAX1)))
```

RULE330: (forall i [indices-of S] [... (in e [indices-of S]) ...]) ->
(forall i [indices-of S] [... (not (after e i)) ...]),
where e is an element of a sequence starting with (indices-of S)

```
(FORALL P1 [INDICES-OF CARDS-PLAYED1]
   [=> (IN ME [INDICES-OF CARDS-PLAYED1])
       (AND [IN (CARDS-PLAYED1 ME)
                (CARDS-IN-SUIT-LED CARDS-PLAYED1)]
            [=> (= (SUIT-OF (CARDS-PLAYED1 P1)) (SUIT-LED))
                (NO. (HIGHER (CARDS-PLAYED1 P1)
                             (CARDS-PLAYED1 ME)))])])
2:82 --- [by RULE330] --->
(FORALL P1 [INDICES-OF CARDS-PLAYED1]
   [=> (NOT (AFTER ME P1))
       (AND [IN (CARDS-PLAYED1 ME)
                (CARDS-IN-SUIT-LED CARDS-PLAYED1)]
            [=> (= (SUIT-OF (CARDS-PLAYED1 P1)) (SUIT-LED))
                (NOT (HIGHER (CARDS-PLAYED1 P1)
                             (CARDS-PLAYED1 ME)))])])
```

5.10.3. Functions as Functionals

In FOO's LISP-like representation, the arguments e_i of an expression (f e_1 ... e_n) may be functions, and the value of the expression may be a function. Such a mapping f from functions to functions is called a *functional*. Functions are sometimes treated as functionals and *vice versa*. For example, the same boolean operator AND is used both as a *function* on truth values and as a *functional* on predicates, under the obvious definition:

(and [lambda (x) P_x] [lambda (y) R_y]) = (lambda (x) (and P_x R_x))

The advantage of treating functions and functionals interchangeably is increased generality: both can be manipulated by the same rules. Essentially, operators like AND are treated as *generic operators*, and knowledge about them is encoded once, under the generic form, rather than separately for each variant corresponding to different argument types. Moreover, this somewhat ambiguous usage saves steps. For example, propositions (representing boolean values) can be conjoined directly with predicates (*functions* onto boolean values) in a mixed notation, allowing expressions like (and [= e_1 e_2] [lambda (x) P_x]). The alternative would require longer expressions, and presumably more steps to construct and manipulate them.

A function f in an expression (f ... [g e_1 ... e_n] ...) can be *functionalized*, *i.e.*, converted into a functional on g, by restructuring the expression as ([f ... g ...] e_1 ... e_n), as illustrated in DERIV14:

```
     (OR [HIGHER (NEXT NOTE) (CLIMAX CANTUS-1)]
         [= (NEXT NOTE) (CLIMAX CANTUS-1)])
14:48 --- [functionalize OR by RULE297] --->
     ([OR HIGHER =] (NEXT NOTE) (CLIMAX CANTUS-1))
```

```
(NOT [(OR HIGHER =) (NEXT NOTE) (CLIMAX CANTUS-1)])
14:49 --- [functionalize NOT by RULE298] --->
([NOT (OR HIGHER =)] (NEXT NOTE) (CLIMAX CANTUS-1))

 (NOT (OR HIGHER =))
14:50 --- [RECOGNIZE by RULE299] --->
 LOWER
```

Here the functions OR and NOT are functionalized respectively by

RULE297: (R [P e_1 e_2] [P' e_1 e_2]) -> ([R P P'] e_1 e_2) -- *functionalize boolean operator R*

RULE298: (not [P ...]) -> ([not P] ...) -- *functionalize* NOT

This makes it possible to merge two cases in terms of a single predicate:

```
(NOT (OR [HIGHER (NEXT NOTE) (CLIMAX CANTUS-1)]
         [= (NEXT NOTE) (CLIMAX CANTUS-1)]))
14:48-50 --- [functionalize and recognize] --->
(LOWER (NEXT NOTE) (CLIMAX CANTUS--1))
```

This kind of transformation is useful in dealing with *fluents* -- entities that change over time -- represented in FOO as unary functions. Two other rules for fluents are illustrated below:

RULE305: (g (f t)) -> ([g f] t), where f is a fluent -- *functionalize g*

```
(NEXT (NOTE T1))
14:84 --- [functionalize NOTE by RULE305] --->
([NEXT NOTE] T1)
```

RULE393: (lambda (t) [g ... (f ... t ...) ...]) -> (g ... [lambda (t) (f ... t ...)] ...) -- *functionalize g*

```
(LAMBDA (J)
   [APPEND (PREFIX CANTUS J) (LIST (NTH CANTUS (+ J 1)))])
14:13 --- [functionalize APPEND by RULE393] --->
(APPEND [LAMBDA (J) (PREFIX CANTUS J)]
        [LAMBDA (J) (LIST (NTH CANTUS (+ J 1)))])

(LAMBDA (J) [LIST (NOTE (+ J 1))])
14:16 --- [functionalize LIST by RULE393] --->
(LIST [LAMBDA (J) (NOTE (+ J 1))])
```

The inverse transformation *defunctionalizes* a functional on f by applying f to an argument:

RULE273: ([g ... f ...] t) -> (g ... [f t] ...), for every fluent f in (g ...) -- *defunctionalize g*

This is illustrated in DERIV14:

```
([OR [AND [= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1]
          [LOWER (NEXT NOTE) (CLIMAX CANTUS-1)]]
     [HIGHER (NEXT NOTE) (CLIMAX CANTUS-1)])
 T1)
14:82 --- [defunctionalize OR by RULE273] --->
(OR [AND [= (#OCCURRENCES (CLIMAX [CANTUS-1 T1]) [CANTUS-1 T1]) 1]
         [LOWER (NEXT [NOTE T1]) (CLIMAX [CANTUS-1 T1])]]
    [HIGHER (NEXT [NOTE T1]) (CLIMAX [CANTUS-1 T1])])
```

5.10.4. Restructuring: Summary

Restructuring transformations help put an expression (f e_1 ... e_n) in the *syntactic form* required by a given operationalization or analysis method. They fall into three classes:

1. *Function transposition* exchanges f with a function g occurring in e_i.
   a. *Distribution*, illustrated by some rules for *propagating splits*, generates more than one copy of f.
   b. *Collection*, used in *merging cases*, combines multiple copies of g into one.
   c. Transposing a single instance of f with a single instance of g *embeds* f and *extracts* g, and helps apply one concept to another by removing an intervening function.
2. *Transfer* moves information from an argument e_i to another argument e_j.
   a. A *change of variable* can be introduced in a quantified expression (Q x S P_x) by transferring information between S and P.
3. *Functionals* -- functions from functions to functions -- can represent entities that depend on *fluents* (functions of time).
   a. Moving information between f and e_i *functionalizes* f -- converts it into a function -- by changing a subexpression (g ...) of e_i to a function g.
   b. Conversely, changing the function g to an expression (g ...) *defunctionalizes* the functional f.

This section discusses restructuring as a syntactic problem-solving tactic, since that is its purpose in FOO. Alternatively, one could classify a restructuring transformation in terms of its effect on the semantics of the expression to which it's applied. This viewpoint involves some mathematical concepts of *invariance*. For example, distribution and collection preserve semantic equivalence when the appropriate *distributive law* holds, and a transposition (f (g e)) -> (g (f e)) preserves equivalence if the operators f and g *commute*. Many of FOO's restructuring transformations do not always preserve equivalence, and it can be far from straightforward to characterize concisely the class of expressions to which a given restructuring rule can be applied without a change of meaning. However, such rules are useful as *heuristic transformations* when they give *empirically valid* results.

5.11. Knowledge Used in Analysis

In (§5.1), FOO's analysis methods are classified based on how they are used. Alternatively, the rules could be classified according to the knowledge they are based on; for instance, rules about sets could be discussed as a group. However, such an organization would reveal little about how the rules are used. Other criteria that might be used to classify analysis methods include:

1. *Kinds of knowledge* needed to apply the method
   a. *Syntactic* knowledge
      Used to match rules against expressions (§B.3)
   b. *Conceptual* knowledge (§1.3)
      Semantic network used in intersection search (§5.6)
      Concept definitions used in elaboration and recognition (§1.3.1)
      Is-a hierarchy used in matching rules (§6.1)
      Relations like OPPOSITE-OF used in rephrasing (§6.1.4)
      Homomorphisms used in function transposition (§6.1.2)
   c. *Procedural* knowledge
      Search algorithm used in intersection search (§5.6)
      Procedures used in systematic evaluation (§5.2.2)
   d. *Logical* knowledge (§5)
      Analysis required to test for special case (§5.9.1)
2. *Semantic relationship* between e and e' in transformation e -> e'
   a. *Equivalence* (valid rules) -- e = e'
      Special-case reduction (§5.9.1)
         Check sufficient condition (§5.9.1.1)
         Check necessary condition (§5.9.1.2)
         Heuristic equivalence-preserving reduction (§5.9.2.1)
      Simplification (§5.2)
      Translation (§5.8)
   b. *Sufficient condition* (sound rules) -- e' => e (§5.9.2.2)

Relevant implication (§5.9.2.2.1)

Partial matching (§5.3.3)

Try example (§5.4.4)

Restrict form of variable (§5.4.2)

c. *Necessary condition* -- e => e' (§5.9.2.3)

Solve equality for value of variable (§5.4.1)

d. *Non-equivalence* -- e ≠ e' (§5.9.2.4)

Defer time frame for achieving goal (§5.9.2.4)

However, these criteria distinguish rather superficially (§5.9.3) between analysis performed in testing a rule condition (§5.9.1) and analysis performed after the rule is applied (§5.9.2). For example, consider two rules:

1. e -> T if C -- *preserves equivalence; verifying the condition C requires analysis* (§5.9.1.1)
2. e -> C -- *transforms e into a sufficient condition C without analysis* (§5.9.2.2)

These rules are almost identical, but would be classified differently with respect to the semantic relationship between left- and right-hand sides: the first rule is valid, while the second is merely sound. Moreover, the distinction breaks down in practice since FOO has mechanisms (§5.9.3.2) for treating the first rule as if it were the second by reducing C to an assumption (§5.9.3.1), restriction (§5.9.3.2.2), or subgoal (§5.9.3.2.1) (§5.9.3.2.3).

The next chapter examines in greater depth the ways in which different forms of *conceptual* knowledge are used in FOO.

Chapter 6
Applications of Conceptual Knowledge

Previous chapters have described FOO's knowledge of *methods* for operationalization (§2) (§3), reformulation (§4), and analysis (§5). These methods draw on knowledge of *concepts*, both *domain-specific* concepts like TRICK, and *general* concepts like HOMOMORPHISM. The various representations used to encode such conceptual knowledge in FOO were described earlier (§1.3) and are illustrated below:

1. Concept definitions, *e.g.*,
   `TAKE-POINTS = (LAMBDA (P) [FOR-SOME C (POINT-CARDS) (TAKE P C)])`
2. Connections in an is-a hierarchy, *e.g.*,
   TRICK is-a ACTION
3. Semantic relations, *e.g.*,
   OPPOSITE of HIGH is LOW
4. Domain-specific "fact" rules, *e.g.*,
   RULE376: (# (cards-in-suit s)) -> 13

This chapter discusses the various ways such knowledge is *used* in the derivations:

1. Match a general rule to a specific concept (§6.1).
2. Expand the definition of a domain concept (§6.2).
3. Recognize an expression as a familiar concept (§6.3).
4. Apply a rule that exploits a known or assumed property of the task domain (§6.4).
5. Find a domain concept satisfying a given test (§6.5).

(§6.6) discusses additional knowledge, missing in FOO, about the semantic appropriateness (validity and soundness) of applying various transformation rules. Finally, (§6.7) summarizes the ways conceptual knowledge is represented and used in FOO.

6.1. Matching General Rules

The semantic relations encoded in FOO (§1.3.2) are primarily used to make contact between general concepts used in rules and more specific concepts used to express and reason about advice. For example, the general concept of reflexive relations is used in a rule for partial matching (§5.3):

RULE43: (R [f e_1 ... e_n] [f e_1' ... e_n']) -> (and [= e_1 e_1'] ... [= e_n e_n']), where R is reflexive

In DERIV5, this rule is applied with R = DURING:

```
(DURING [TAKE (TRICK-WINNER) C1] [TAKE ME QS])
5:12 --- [REDUCE by RULE43] --->
(AND [= (TRICK-WINNER) ME] [= C1 QS])
```

This transformation is justified (*i.e.*, logically sound) because DURING is-a REFLEXIVE. The same rule applies to other relations as well, *e.g.*, => and =:

```
(=> [MOVE C4 (HAND P1) POT] [MOVE C3 (HAND (QSO)) LOC3])
3:52 --- [REDUCE by RULE43] --->
(AND [= C4 C3] [= (HAND P1) (HAND (QSO))] [= POT LOC3])

                [= (HAND P1) (HAND (QSO))]
3:55            --- [REDUCE by RULE43] --->
                (AND [= P1 (QSO)])
```

FOO's rule matcher searches the is-a hierarchy to determine whether a component of an expression matches the corresponding component of a rule. It matches the expressions above by finding the direct connection => is-a REFLEXIVE and the path = is-a EQUIVALENCE is-a REFLEXIVE.

FOO also uses stored semantic relations to:

Recognize instances of specialized concepts used in rules (§6.1.1).

Exploit homomorphic relationships (§6.1.2).

Simplify expressions involving identity elements (§6.1.3).

Find connections between related adjectives (§6.1.4).

6.1.1. Recognizing Special Cases

The is-a hierarchy can be used to detect *special cases*. For example, in DERIV13 a reduction is made based on the knowledge that (#OCCURRENCES ...) must be non-negative:

```
(=< (#OCCURRENCES CLIMAX1 PROJECT1) 0)
13:111 --- [RECOGNIZE by RULE392] --->
(= (#OCCURRENCES CLIMAX1 PROJECT1) 0)
```

This transformation is made by

RULE392: (=< e 0) -> (= e 0) if e is non-negative

To infer that (#OCCURRENCES ...) is non-negative, FOO first retrieves the definition

```
#OCCURRENCES = (LAMBDA (N C) (# (SET-OF X C (= X N))))
```

FOO then finds the path `# is-a NON-NEGATIVE-INTEGER is-a NON-NEGATIVE`. This justifies applying RULE392, which is only valid in the special case where e is non-negative.

6.1.2. Exploiting Homomorphisms

Semantic relations other than "is-a" are also used in applying general rules to specific concepts. One such rule encodes knowledge about *homomorphisms*:

RULE284: (f ... [g e_1 ... e_n] ...) -> (g' [f ... e_1 ...] ... [f ... e_n ...]),
where (lambda (x) (f ... x ...)) is a homomorphism, and g and g' correspond to + and +' in the homomorphism condition f(x + y) = f(x) +' f(y)

To apply RULE284 to an expression (f ...) where `f is-a HOMOMORPHISM`, FOO uses the stored relation `ADD of f is g'`. Each example below is preceded by the relation used.

```
ADD of CHOICE-SEQ-OF is APPEND (g is SCENARIO)

(CHOICE-SEQ-OF [SCENARIO (EACH P1 (PLAYERS) (PLAY-CARD P1))
                         (TAKE-TRICK (TRICK-WINNER))])
2:6 --- [by RULE284] --->
(APPEND [CHOICE-SEQ-OF (EACH P1 (PLAYERS) (PLAY-CARD P1))]
        [CHOICE-SEQ-OF (TAKE-TRICK (TRICK-WINNER))])

ADD of HAVE-POINTS is OR (g is APPEND)

(HAVE-POINTS [APPEND CARDS-PLAYED1 (LIST C21)])
2:102 --- [DISTRIBUTE by RULE284] --->
(OR [HAVE-POINTS CARDS-PLAYED1] [HAVE-POINTS (LIST C21)])

ADD of IN is OR (g is UNION)

(IN (LOC QS)
    [UNION (SET DECK POT HOLE (HAND ME)) (PILES (PLAYERS))])
4:31 --- [DISTRIBUTE by RULE284] --->
(OR [IN (LOC QS) (SET DECK POT HOLE (HAND ME))]
    [IN (LOC QS) (PILES (PLAYERS))])

ADD of DURING is OR (g is SCENARIO)
```

```
(DURING [SCENARIO (EACH P1 (PLAYERS) (PLAY-CARD P1)
                  (TAKE-TRICK (TRICK-WINNER))]
        (TAKE-POINTS ME))
6:4 --- [DISTRIBUTE by RULE284] --->
(OR [DURING (EACH P1 (PLAYERS) (PLAY-CARD P1)) (TAKE-POINTS ME)]
    [DURING (TAKE-TRICK (TRICK-WINNER)) (TAKE-POINTS ME)])

ADD of #OCCURRENCES is + (g is APPEND)

(#OCCURRENCES CLIMAX1 [APPEND PROJECT1 (LIST NOTE3)])
13:106 --- [DISTRIBUTE by RULE284] --->
(+ [#OCCURRENCES CLIMAX1 PROJECT1]
   [#OCCURRENCES CLIMAX1 (LIST NOTE3)])
```

6.1.3. Identity Elements

Two general simplification rules about *identity elements* of additive operators are applied to a variety of expressions based on the stored relation IDENTITY of C is i:

RULE341: (C i e) -> e, where C is a commutative operator and i is its identity element

RULE343: (C e i) -> e, where C is a commutative operator and i is its identity element

A few of the many expressions simplified by these rules are shown below, preceded by the relevant relations:

```
IDENTITY of AND is T

3:19 (AND T [LEGAL (QSO) QS]) ---> (LEGAL (QSO) QS)

IDENTITY of OR is NIL

5:9 (OR NIL [DURING (TAKE-TRICK (TRICK-WINNER)) (TAKE ME QS)])

IDENTITY of D* is INCREASING

9:48 (D* INCREASING [DEPENDENCE (PR-DISJOINT-FORMULA #H #S #U) #S])

IDENTITY of D+ is NIL

9:53 (D+ [DEPENDENCE (#CHOOSE (- #U #S) #H) #S] NIL)

IDENTITY of + is 0

14:26 (+ [#OCCURENCES (CLIMAX CANTUS-1) CANTUS-1] 0)
```

6.1.4. Related Adjectives

FOO's semantic relations are also used to relate different forms of *adjectives*. For instance, the adjective LOW, a unary predicate, has LOWER as its comparative form, LOWEST as its superlative, and HIGH as its opposite. These relations are exploited using the rules listed below with examples of their use and the relations involved.

RULE153: $(R\ e_1\ e_2) \rightarrow (R'\ e_1\ e_2)$, where OPPOSITE of R is R'

```
OPPOSITE of HIGHER is LOWER

(HIGHER X1 (CARD-OF ME))
7:5 --- [by RULE153] --->
(LOWER (CARD-OF ME) X1)

OPPOSITE of >= is =<

(>= 1 (#OCCURRENCES CLIMAX1 PROJECT1))
13:87 --- [by RULE153] --->
(=< (#OCCURRENCES CLIMAX1 PROJECT1) 1)
```

RULE154: (R e x) -> (P e), where PREDICATE of R is P

```
PREDICATE of LOWER is LOW

(LOWER (CARD-OF ME) X1)
7:6 --- [REDUCE by RULE154] --->
(LOW (CARD-OF ME))
```

RULE277: (change (f S)) -> (R [f (change S)] [f S]), where COMPARATIVE of f is R

```
COMPARATIVE of HIGHEST is HIGHER

(CHANGE (HIGHEST CANTUS-1))
14:42 --- [REDUCE by RULE277] --->
(HIGHER [HIGHEST (CHANGE CANTUS-1)] [HIGHEST CANTUS-1])
```

RULE300: (next (f S)) -> (f (change S)) if (R [f (change S)] [f S]),
where COMPARATIVE of f is R

```
COMPARATIVE of HIGHEST is HIGHER

(NEXT (HIGHEST CANTUS-1))
14:55-57 --- [REDUCE by RULE300, analysis] --->
(HIGHEST (CHANGE CANTUS-1))
since (HIGHER [HIGHEST (CHANGE CANTUS-1)] [HIGHEST CANTUS-1])
```

RULE304: (in e S) -> nil if (R e [f S]),
where R is some ordering and SUPERLATIVE of R is f

```
SUPERLATIVE of HIGHER is HIGHEST
```

```
(IN (NEXT NOTE) CANTUS-1)
14:72-76 --- [REDUCE by RULE304, analysis] --->
NIL
since (HIGHER [NEXT NOTE] [HIGHEST CANTUS-1])
```

RULE367: (not (P e)) -> (P' e), where `OPPOSITE of P is P'`

```
OPPOSITE of LOW is HIGH

(NOT (LOW (CARD-OF ME)))
7:8 --- [by RULE367] --->
(HIGH (CARD-OF ME))
```

6.2. Expanding Concept Definitions

Expanding the definition of a concept helps bring detailed lower-level (general) knowledge about it to bear on the problem at hand. Such *elaboration* (§5.8.2) can help map a problem to a known method. In DERIV9, expanding the definition of `VOID` produces a predicate on the unevaluable set `(CARDS-IN-HAND P0)` and suggests using the disjoint subsets method (§2.3):

```
(VOID P0 S0)
9:1 --- [ELABORATE by RULE124] --->
(NOT (EXISTS C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0)))
9:2-26 --- [by RULE189, analysis] --->
(PR-DISJOINT (CARD-IN-HAND P0) (CARDS-IN-SUIT S0) (CARDS))
```

If a concept is defined in terms of an aggregate of components, such as a conjunction, disjunction, or sequence, plugging its definition into a problem may help split it into separately solvable subproblems. This use of elaboration is discussed at length elsewhere (§5.7.4).

6.3. Recognizing Concept Definitions

Recognizing an expression as an instance of a defined concept (§5.8.3) can also help map the expression to a known method, either a rule or a previously generated operationalization (§6.3.1). In simple cases, recognition can be performed by substituting a concept name for a sub-expression in the problem; this corresponds to the *folding* transformation in [Darlington 76]. More complicated cases involve solving a reformulation equation (§4) for an appropriate function (§6.3.2).

An example of recognition occurs in DERIV12, where an intersection search rule is used to show that voids can't be undone:

RULE57: (during s e) -> nil if no sub-events of s are abstractions of e

To use the rule, c must be expressed in terms of a *known action* concept more specific than MOVE. This is accomplished by recognizing (UNDO (VOID P0 S0)) in terms of GET-CARD:

```
(DURING (ROUND-IN-PROGRESS)
        (UNDO (VOID P0 S0)))
12:2-9  --- [by analysis] --->
(DURING (ROUND-IN-PROGRESS)
        (EXISTS C1 (CARDS) (AND [MOVE C1 LOC1 (HAND P0)]
                                [IN-SUIT C1 S0])))
12:10                           --- [RECOGNIZE by RULE123] --->
(DURING (ROUND-IN-PROGRESS)
        (EXISTS C1 (CARDS) (AND [GET-CARD P0 C1 LOC1]
                                [IN-SUIT C1 S0])))
12:11 --- [COMPUTE by RULE57] --->
NIL
```

Similarly, recognition can help reformulate an expression in terms of an *executable action*. In DERIV8, an abstract plan for getting void in a suit is made concrete by recognizing PLAY:

```
(ACHIEVE (VOID ME S0))
8:1-10 --- [by RULE6, analysis] --->
(UNTIL (VOID ME S0)
       (ACHIEVE (AND [MOVE C1 (HAND ME) LOC1] [IN-SUIT C1 S0])))
8:11                --- [RECOGNIZE by RULE123] --->
(UNTIL (VOID ME S0)
       (ACHIEVE (AND [PLAY ME C1] [IN-SUIT C1 S0])))
8:13            --- [RECOGNIZE by RULE123] --->
(UNTIL (VOID ME S0)
       (ACHIEVE (PLAY-SUIT ME S0)))
```

The concept PLAY-SUIT is recognized simply for convenience, to shorten the solution; this doesn't make the plan any more operational. A caveat to this statement has to do with a bit of lazy notation used in the derivation. The lack of an explicit existential quantifier for the variable C1 comes from using

RULE7: (remove-1-from (set-of x S P_x)) -> (undo (and [in x S] P_x))

```
(REMOVE-1-FROM (SET-OF C1 (CARDS-IN-HAND ME) (IN-SUIT C1 S0)))
8:4 --- [REDUCE by RULE7] --->
(UNDO (AND [IN C1 (CARDS-IN-HAND ME)] [IN-SUIT C1 S0]))
```

The alternative is to use the more precise

RULE7a: (remove-1-from (set-of x S P_x)) -> (for-some x S (undo (and [in x S] P_x)))

```
(REMOVE-1-FROM (SET-OF C1 (CARDS-IN-HAND ME) (IN-SUIT C1 S0)))
8:4' --- [REDUCE by RULE7a] --->
(FOR-SOME C1 (CARDS-IN-HAND ME)
   (UNDO (AND [IN C1 (CARDS-IN-HAND ME)] [IN-SUIT C1 S0])))
```

This would produce the expression

```
(FOR-SOME C1 (CARDS-IN-HAND ME)
   (AND [PLAY ME C1] [IN-SUIT C1 S0]))
```

This expression is executable given the ability to scan through (CARDS-IN-HAND ME), test the suit of each card, choose one in suit S0, and play it. Moreover, it would match a proper definition of PLAY-SUIT, as opposed to the *ad hoc* one used above, with its implicitly quantified variable ?C:

```
PLAY-SUIT = (LAMBDA (P S) (AND [PLAY ME ?C] [IN-SUIT ?C S]))
```

The point is that the inclusion of PLAY-SUIT in the knowledge base was not necessary for deriving an operational solution to this problem; it was only included to provide a convenient name for a derived concept.

6.3.1. Recognition and Learning

In the examples above, recognition was used to map problems onto general methods (transformation rules) or runtime capabilities (executable actions and permissible observations). Recognition could also be used to map problems onto previously derived operationalizations. For example, suppose the concept TRICK-HAS-POINTS has been operationalized by deriving methods for predicting whether a trick will have points. These methods could be retrieved when a subsequent instance of TRICK-HAS-POINTS was recognized, as in DERIV6:

```
(AVOID-TAKING-POINTS (TRICK))

6:1-20 --- [by analysis] --->
(ACHIEVE (NOT (AND [= (TRICK-WINNER) ME]
                   [EXISTS C3 (CARDS-PLAYED) (HAS-POINTS C3)])))

6:21               --- [RECOGNIZE by RULE123] --->
(ACHIEVE (NOT (AND [= (TRICK-WINNER) ME]
                   [HAVE-POINTS (CARDS-PLAYED)])))

6:22               --- [RECOGNIZE by RULE123] --->
(ACHIEVE (NOT (AND [= (TRICK-WINNER) ME] [TRICK-HAS-POINTS])))
```

Reformulating AVOID-TAKING-POINTS in terms of a conjunction and recognizing the second conjunct as TRICK-HAS-POINTS would make it possible to apply any previously derived methods for evaluating TRICK-HAS-POINTS. In particular, when the methods predicted that a trick would not have points, any card could be played without fear of taking points.

As implemented, FOO is used to operationalize expressions independently of each other. However, retrieving solutions of previously solved subproblems would be necessary in a practical operationalizer to avoid the expense of solving every problem from scratch. This simple technique proved useful in the Logic Theorist program, which retrieved previously generated proofs to prove parts of new theorems [Newell 57]. The role of recognition in such a process is to relate new problems to previously solved ones, thereby broadening the range of application of stored solutions.

6.3.2. Recognition by Matching versus Recognition by Analysis

Some of FOO's concept definitions can be criticized for the serendipity with which they match synthesized expressions recognized in terms of those concepts. RULE123, the rule for recognizing an expression as a known concept, is quite primitive: it just tests for a literal match between expression and concept definition, allowing for variable substitutions. To illustrate, consider the recognition transformation

```
(FOR-SOME C1 (CARDS-IN-HAND ME)
   (AND [PLAY ME C1] [IN-SUIT C1 S0]))
--- [RECOGNIZE] --->
(PLAY-SUIT ME S0)
```

RULE123 can be used to make this transformation if PLAY-SUIT is defined as

```
PLAY-SUIT =
  (LAMBDA (P S)
     (FOR-SOME X (CARDS-IN-HAND ME)
        (AND [PLAY P X] [IN-SUIT X S])))
```

This would be done by identifying P, S, and X with ME, S0, and C1, respectively. However, RULE123 would not be able to make the same transformation given the almost identical definition

```
PLAY-SUIT =
  (LAMBDA (P S)
     (FOR-SOME X (CARDS)
        (AND [PLAY P X] [IN-SUIT X S])))
```

The problem here is that (CARDS-IN-HAND ME) doesn't match (CARDS). In short, RULE123 is not robust with respect to variations in the definition of the concept to be recognized.

Such variations could be allowed by using *analysis* for recognition. An expression ($f\,e_1 \ldots e_n$) could be matched to a definition (lambda ($x_1 \ldots x_n$) <body>) by solving for $x_1 \ldots x_n$ in the equation

($f\,e_1 \ldots e_n$) = <body>

In the above example, this would mean solving for P and S in

```
(= [FOR-SOME C1 (CARDS-IN-HAND ME)
      (AND [PLAY ME C1] [IN-SUIT C1 S0])]
   [FOR-SOME X (CARDS)
      (AND [PLAY P X] [IN-SUIT X S])])
```

This might involve showing that a card must be in player P's hand in order for P to play it, *i.e.*,

```
(=> [PLAY P X] [IN X (CARDS-IN-HAND P)])
```

Using analysis in recognition would increase the range of expressions recognizable as a given concept, at the cost of slowing down the recognition process in the easy cases handled by the simple matching algorithm. As implemented, RULE123 recognizes an expression (f ...) by first retrieving each concept whose definition has the form (lambda (...) (f ...)); all such concepts are indexed under f. The expression is then matched against each retrieved definition to find the concept that fits. The total process is reasonably fast. Recognition-by-analysis would be much slower, especially if it tested concepts other than those with definitions of the form (lambda (...) (f ...)). (This might be necessary to tolerate differences between the stored definition and the expression to be recognized.) To simulate the robustness of recognition-by-analysis without sacrificing the speed of recognition-by-matching, I tailored the definitions of a few concepts to fit synthesized expressions that occurred in the derivations. In most cases this simply shortened an already operational expression.

6.4. Exploiting Properties of the Task Domain

Knowledge about task domain structure is reflected in implicit and explicit *assumptions* made in the various derivations. Examination of these assumptions sheds light on a central question:

What kind of knowledge about the task domain is used in operationalizing advice?

(§6.4.1) examines the central assumption that expressions used in advice and concept definitions are well-defined. (§6.4.2) describes some constructs containing embedded assumptions. (§6.4.3) considers definitions as assertions about the task domain. (§6.4.4) discusses explicit assumptions about the task domain made in some of the derivations.

6.4.1. The Well-definedness Assumption

If the knowledge base is a consistent *model* of the task domain, and advice about the task is meaningful, certain properties of the domain are captured by a general *well-definedness assumption*:

Assume any expression in a definition or heuristic will be well-defined whenever it is used.

In particular, expressions of the form (the x S P_x) are only well-defined when S contains an element satisfying P and that element is unique in S. For instance, consider the definitions

```
HIGHEST =
  (LAMBDA (S)
     (THE C S
        (FORALL X S (NOT (HIGHER X C)))))

(WINNING-CARD)
= (HIGHEST-IN-SUIT-LED (CARDS-PLAYED))
= (HIGHEST (CARDS-IN-SUIT-LED (CARDS-PLAYED)))
```

The function HIGHEST is only well-defined when S has a unique highest element: thus the concept (WINNING-CARD) only makes sense if (CARDS-IN-SUIT-LED (CARDS-PLAYED)) contains a card higher in the card ordering than all the others.

The fact that (WINNING-CARD) is well-defined reflects, among other things, that Hearts is played with a single deck of cards -- if two of the same card were played, which would win the trick? Actually, there are situations in which (WINNING-CARD) is undefined, for example during the dealing, passing, and scoring phases of the game, when (CARDS-PLAYED) is undefined or empty. However, the *(WINNING-CARD) concept is not used during these phases.* The well-definedness assumption requires only that a concept be well-defined *whenever it is used.* More precisely, if a task definition is viewed as an executable program, then an expression is *used* at those points in the execution of the program when it's evaluated.

Note also that the assumption talks only about *well-definedness*; it says nothing about *evaluability*. For example, consider the predicate

```
OUT = (LAMBDA (C) (EXISTS P (OPPONENTS ME) (HAS P C)))
```

This predicate is well-defined, but cannot be computed in the straightforward way by player ME without violating the rules of the game (no peeking at opponents' hands). Even using the pigeon-hole principle (§2.2), there is no perfect way to decide if a card is out, since it may have been the hole card.

6.4.2. Embedded Assertions

An assertion C is said to be embedded in an expression e if C must be true in order for e to make sense. For example, any expression of the form (the x S P_x) contains the embedded assertion (exists-unique x S P_x). Similarly, the construct (partition S_1 ... S_n) contains the embedded assertion (disjoint S_1 ... S_n). This assertion is made into an explicit assumption by

RULE15: (partition $S_1 \ldots S_n$) -> (union $S_1 \ldots S_n$); assume (disjoint $S_1 \ldots S_n$)

In its explicit form, this assumption can be used by

RULE19: (diff [join $S_1 \ldots S \ldots S_n$] S) -> (join $S_1 \ldots S_n$) if (disjoint $S_1 \ldots S_n$)

This is illustrated in DERIV4:

```
(DIFF [PARTITION (HANDS (PLAYERS))
                 (PILES (PLAYERS))
                 (SET DECK POT HOLE)]
      [PROJECT HAND (OPPONENTS ME)])

4:10 --- [ELABORATE by RULE15] --->
(DIFF [UNION (HANDS (PLAYERS))
             (PILES (PLAYERS))
             (SET DECK POT HOLE)]
      [PROJECT HAND (OPPONENTS ME)])
assume (DISJOINT (HANDS (PLAYERS))
                 (PILES (PLAYERS))
                 (SET DECK POT HOLE))

4:11-15 --- [by analysis] --->
(UNION (DIFF [UNION (HANDS (PLAYERS))
                    (PILES (PLAYERS))
                    (SET DECK POT HOLE)]
             [HANDS (PLAYERS)])
       (INTERSECT [UNION (HANDS (PLAYERS))
                         (PILES (PLAYERS))
                         (SET DECK POT HOLE)]
                  [SET (HAND ME)]))

4:16-18 --- [by RULE19, previous assumption] --->
(UNION [UNION (HANDS (PLAYERS))
              (PILES (PLAYERS))
              (SET DECK POT HOLE)]
       (INTERSECT [UNION (HANDS (PLAYERS))
                         (PILES (PLAYERS))
                         (SET DECK POT HOLE)]
                  [SET (HAND ME)]))
```

In this example, the disjointness assertion follows from the assumption that (partition ...) is well-defined, and models the fact that hands, piles, deck, pot, and hole are distinct locations. This property is exploited by the special-case reduction rule RULE19, whose condition tests for disjointness. Thus the implicit disjointness assumption and the way it is used to satisfy a special-case test are both quite clear. For the construct (the x S P_x), the implicit unique-existence assumption is clear, but how it's used is not.

6.4.3. Definitions as Assertions about a Modelled Domain

One answer to the question of how such assumptions are used views the knowledge base as a *model of external phenomena.* Assumptions of well-definedness reflect *predictions* about these phenomena, *e.g.*, "at the end of every trick, the pot will contain exactly one highest card in the suit led." Properties of the task domain are exploited to *simplify the task description.* This affects operationalization insofar as simpler task descriptions are easier to reason about. The definition of a trick wouldn't be as simple if it failed to exploit the uniqueness of the trick-winner. For example, the event of the winner taking the trick could not be defined as

```
(TAKE-TRICK [THE P (PLAYERS)
               (FORALL C (CARDS-PLAYED)
                  (NOT (HIGHER C (CARD-OF P))))])
```

An extra quantifier would be needed to deal with a possibly non-unique "player of the highest card":

```
(FOR-SOME P [SET-OF Q (PLAYERS)
               (FORALL C (CARDS-PLAYED)
                  (NOT (HIGHER C (CARD-OF Q))))]
   (TAKE-TRICK P))
```

Both expressions describe the same event, but the second is presumably harder to reason about.

If the knowledge base is a model of external pheonomena, then by the well-definedness assumption, every function in the knowledge base represents an assertion about the domain: namely that the external entity denoted by the function is uniquely determined by the values of its arguments plus the values of any fluents in its definition. (The latter rather gaping loophole could be closed by replacing those fluents with explicit arguments.) For example, consider the definition

```
(WINNING-CARD) = (HIGHEST (CARDS-IN-SUIT-LED (CARDS-PLAYED)))
```

This definition implicitly asserts that the winning card (the *external entity* modelled by `WINNING-CARD`) is uniquely determined by the cards played in the trick, independent of, say, what cards have been taken so far. This property of the game makes it simpler to describe (and reason about) than if the winning card depended on the locations of all 52 cards, the current score, and the day of the week. The discovery of such *invariances* is a central problem in experimental science [Langley 79].

6.4.4. Explicit Assumptions about the Task Domain

Some of the derivations involve *explicit* assumptions about the task domain generated in the course of applying various transformations. Such assumptions are illustrated in DERIV3, where the goal is to construct a plan for flushing the Queen of spades. A subgoal is to make it legal for the player (QSO) who has the Queen to play it. This happens if player (QSO) has the lead, or if spades are led. However, the first case violates the subgoal of constraining player (QSO)'s options. FOO doesn't know about the flexibility associated with having the lead; I *simulated* this knowledge by assumption:

RULE342: (= c e) -> T by assumption, where c is a constant

```
(= S (SUIT-LED))
3:26 --- [ASSUME by RULE342] --->
T
assume (SUIT-LED) = S
```

RULE349: (=> C P) -> T by assuming P false

```
(=> (LEADING (QSO))
    (OR [CAN-LEAD-HEARTS (QSO)] [NEQ (SUIT-OF C3) H]))
3:80 --- [ASSUME by RULE349] --->
T
assume (LEADING (QSO)) = NIL
```

These assumptions helped reduce the goal of flushing the Queen to the goal of making player (QSO) keep playing spades. The fact that these conditions are sufficient to force player (QSO) to play a spade follows from the rules of the game. This deduction was simulated as follows:

RULE351: (achieve P) -> (achieve P_1 ... P_n) if assumptions P_1 ... P_n reduce original goal to P

```
(ACHIEVE (PLAY-SPADE (QSO)))
3:88 --- [REDUCE by RULE351] --->
(ACHIEVE (AND [= (LEADING (QSO)) NIL] [= (SUIT-LED) S]))
```

This reduction led to an operational plan for flushing the Queen by repeatedly leading spades.

Another explicit assumption reflecting a property of the task domain is used in DERIV10:

RULE373: (at c hole) -> nil by assumption

```
(AT X1 HOLE)
10:19 --- [ASSUME by RULE373] --->
NIL
```

The knowledge incorporated in this assumption is that fact that the probability of any given card being the hole card is often low enough to ignore (§1.3.3).

6.5. Knowledge Base Search

A key feature of FOO is the ability to search through the knowledge base of concept definitions for a concept satisfying a specified condition or for a specified relationship between given concepts. Some types of search are based on the assumption that the knowledge base is complete in a certain way (§6.5.1). Another type of search exploits the fact that concepts included in the knowledge base are more likely to be useful than randomly generated expressions (§6.5.2). Knowledge base search can find an action with a specified effect (§6.5.3), or a testable necessary condition for an assertion that cannot be directly evaluated (§6.5.4). The ability to search through the knowledge base implies that simply adding the right concept to the knowledge base may enhance the ability to operationalize (§6.5.5).

6.5.1. Closure-based Search

Some types of search exploit *closed-universe* assumptions. For example, the following rules assume that if an event E can occur during a scenario S, the definition of S mentions E directly or indirectly (§2.4.5.2):

RULE57: (during S E) -> nil if sub-events of S and abstractions of E do not intersect

RULE306: (P ... e ...) -> (HSM with [problem : (P ... e ...)] [choice-seq : (choice-seq-of e)]) if a sequence subevent of e contains a choice event of the form (choose x S E_x)

RULE308: (choice-seq-of S) -> nil if no sub-event of S is a choice event

This assumption permits certain questions to be answered quickly by intersection search through the knowledge base (§5.6):

```
(DURING [EACH P1 (PLAYERS) (PLAY-CARD P1)] [TAKE ME QS])
5:8 --- [COMPUTE by RULE57] --->
NIL

(DURING [EACH P1 (PLAYERS) (PLAY-CARD P1)] [TAKE-POINTS ME])
6:5 --- [COMPUTE by RULE57] --->
NIL

(DURING [ROUND-IN-PROGESS]
        [EXISTS P1 (OPPONENTS ME) (GET-CARD P1 X1 LOC1)])
10:10 --- [COMPUTE by RULE57] --->
NIL

(DURING [ROUND-IN-PROGRESS]
        [EXISTS C1 (CARDS)
           (AND [GET-CARD P0 C1 LOC1] [IN-SUIT C1 S0])])
12:11 --- [COMPUTE by RULE57] --->
NIL
```

```
(CHOICE-SEQ-OF (TAKE-TRICK (TRICK-WINNER)))
2:7 --- [COMPUTE by RULE308] --->
NIL since no sub-event of TAKE-TRICK is a choice event

(DURING (TRICK) (TAKE-POINTS ME))
2:3 --- [by RULE306] --->
(HSM1 WITH [PROBLEM : (DURING (TRICK) (TAKE-POINTS ME))]
           [CHOICE-SEQ : (CHOICE-SEQ-OF (TRICK))])
since choice event PLAY-CARD is in a sequence sub-event of (TRICK)

(LEGAL-CANTUS! (CANTUS))
13:1 --- [by RULE306] --->
(HSM1 WITH [PROBLEM : (LEGAL-CANTUS! (CANTUS))]
           [CHOICE-SEQ : (CHOICE-SEQ-OF (CANTUS))])
since choice event CHOOSE-NOTE is in sequence event (CANTUS)
```

6.5.2. Plausible Generation of Sufficient Conditions

Other types of knowledge base search exploit the fact that *defined concepts are likely to be useful.* For example, a predicate that someone has bothered to define is likely to be satisfied reasonably often by some combination of task situation components. (Most words are not invented to describe things that seldom arise.) This property motivates the use of knowledge base search for *plausible generation* of concepts satisfying desired properties. For instance, the following two rules try to generate *sufficient conditions* for predicates that can't be evaluated directly:

RULE227: (P ...) -> (=> [R ...] [P ...]) if (R ...) is true and definition of R mentions P

```
(VOID P0 S0)
11:1-6 --- [by RULE227, analysis] --->
(=> [LEGAL P1 (CARD-OF P1)] [VOID P0 S0])
since (LEGAL P1 (CARD-OF P1)) is true for all P1
11:7-18 --- [REDUCE by analysis] --->
(AND [BREAKING-SUIT P0] [= (SUIT-LED) S0] [FOLLOWING P0])
```

RULE319: (P ...) -> add annotation (R ...) is necessary when C,
where definition of R mentions P, and C is result of reducing (=> [P ...] [R ...])

```
(HAS P1 C2)
2:28 --- [by RULE319] --->
(HAS P1 C2) ; [=> (OUT C7) IF (=> [HAS P1 C2] [OUT C7])]
2:29-39     --- [REDUCE by analysis] --->
(HAS P1 C2) ; [=> (OUT C2) IF (IN P1 (OPPONENTS ME))]

(HAS P1 C2)
2:40 --- [by RULE319] --->
(HAS P1 C2) ; [=> (NON-VOID P5) IF (=> [HAS P1 C2] [NON-VOID P5])]
2:41-57     --- [REDUCE by analysis] --->
(HAS P1 C2) ; [=> (NON-VOID P1) IF (IN-SUIT C2 (SUIT-LED))]
```

Both RULE227 and RULE319 search the knowledge base for a predicate R whose definition mentions P directly or indirectly. This requirement that R be *relevant* to P improves the chances that

(=> [R ...] [P ...]) or (=> [P ...] [R ...]) is reducible (by partial matching) to an evaluable expression (§2.6.4.1).

6.5.3. Finding Actions with Specified Effects

A similar use of the knowledge base is to *find an action with a specified effect*. Often this is achieved by *recognizing* an action concept whose definition matches a described change (§2.4). This doesn't always work, for example when the specified effect is that the action leave some condition unchanged. When recognition doesn't work, a less efficient but more robust alternative is a generate-and-test approach: *generate* candidate actions from the knowledge base of defined action concepts and *test* each one by analysis. In practice, this search was partly simulated by hand-selecting the correct concept from the set of defined actions, but this concept was tested using FOO's analysis rules. In the rules below, A is an action selected from the knowledge base:

RULE234: (P ...) -> (was-during [current A] [P ...]) if (not (during [A ...] [undo P]))

```
(VOID P0 S0)
12:1-13 --- [by RULE234, analysis] --->
(WAS-DURING [CURRENT ROUND-IN-PROGRESS] [VOID P0 S0])
since (NOT (DURING [ROUND-IN-PROGRESS] [UNDO (VOID P0 S0)]))
```

RULE254: (change P) -> (A ...) if (=> [A ...] [change P]), where A is an action

```
(UNDO (HAS (QSO) C3))
3:46-61 --- [by RULE254, analysis] --->
(PLAY (QSO) C3)
since (=> [PLAY P1 C4] [UNDO (HAS (QSO) C3)])
if C4 = C3 and P1 = (QSO)
```

RULE356: (P ...) -> (or [was-during [current A] [cause (P ...)]] [before [current A] [P ...]])

```
(AT QS (PILE P3))
4:47 --- [by RULE356] --->
(OR [WAS-DURING [CURRENT ROUND-IN-PROGRESS]
                [CAUSE (AT QS (PILE P3))]]
    [BEFORE [CURRENT ROUND-IN-PROGRESS] [AT QS (PILE P3)]])
```

RULE370: (# (set-of x S P_x)) ->
(- [# (set-of x S (before [current A] P_x))]
[# (set-of x S (was-during [current A] [undo P_x]))])

```
(# (SET-CF X1 (CARDS-IN-SUIT S0) (OUT X1)))
10:4-12 --- [by PULE370, analysis] --->
(- [# (SET-OF X1 (CARDS-IN-SUIT S0)
          (BEFORE [CURRENT ROUND-IN-PROGRESS] [OUT X1]))]
   [# (SET-OF X1 (CARDS-IN-SUIT S0)
          (WAS-DURING [CURRENT ROUND-IN-PROGRESS] [UNDO (OUT X1)]))])
```

6.5.4. Find a Necessary Condition

A similar use of the knowledge base is to find a *necessary condition* for a given expression. A rule that tests a necessary condition for membership in a collection is

RULE304: (in e S) -> nil if (R e [f S]), where R is an ordering and SUPERLATIVE of R is f

```
(IN (NEXT NOTE) CANTUS-1)
14:73-75 --- [by RULE304] --->
NIL since (HIGHER (NEXT NOTE) (HIGHEST CANTUS-1))
```

The domain-specific ordering R, in this case the musical relation HIGHER, is selected from the ordering concepts in the knowledge base.

A rule for refining the disjoint subsets method (§2.3.1.5) searches the knowledge base for a necessary condition P for membership in the set H:

RULE200: (Pr-disjoint H S U) -> (Pr-disjoint H (filter S P) (filter U P)) if (filter H P) = H, where P is a unary predicate selected from the knowledge base of domain concepts

RULE200 refines an estimate of the probability that player P0 is void in suit S0 by finding the concept of a card being OUT (held by an opponent):

```
(PR-DISJOINT (CARDS-IN-HAND P0) (CARDS-IN-SUIT S0) (CARDS))
9:27-40 --- [REFINE by RULE200, analysis] --->
(PR-DISJOINT (CARDS-IN-HAND P0) (CARDS-OUT-IN-SUIT S0) (CARDS-OUT))
since (FILTER (CARDS-IN-HAND P0) OUT) = (CARDS-IN-HAND P0)
assuming (IN P0 (OPPONENTS ME))
```

6.5.5. Having a Concept

An interesting consequence of using methods that search through a knowledge base of domain concepts is that the mere *presence* of the right concept in the knowledge base may greatly improve the ability to operationalize advice by enabling the derivation of solutions not otherwise possible. This phenomenon appears to capture the flavor of a familiar aspect of human thought -- the usefulness of "having" a concept relevant to the problem at hand. For example, my ability to reason about Hearts improved when I was told about the concept of *distribution*. This gave me a powerful reasoning tool; by reformulating a question about a game situation in terms of distribution, I was able to compute the answer easily:

1. I'm pretty sure Mike's got the Jack of diamonds: does he have any other diamonds left?
2. I know there are three diamonds out; they're probably distributed 1-2 or 2-1.

3. There's roughly a 50-50 chance the Jack's his last diamond, so it's worth leading my Ace.

This probability estimate is actually wrong, but it reflects my actual reasoning in a recent game. (Despite my sloppy reasoning, my conclusion was correct: I played the Ace and flushed the Jack.)

Although FOO cannot at present exploit a concept as sophisticated as "distribution" just by knowing its definition (it would be interesting to try to get it to do so on the same example), it was able to refine an estimate of the probability of a void because the concept OUT happened to be defined in the knowledge base, having been used in the advice "if the Queen is out, try to flush it."

6.6. Missing Rule Conditions as Domain Knowledge

The act of applying a rule assumes that doing so is semantically appropriate for the problem at hand, *e.g.*, logically valid or sound. It is not always easy to test mechanically whether a rule is semantically appropriate, or even to formulate precisely what the test should be. Many of FOO's rules are syntactically applicable to expressions to which it would be inappropriate to apply them. Restricting the rules to prevent such inappropriate applications would involve adding extra rule conditions. Such conditions constitute domain knowledge I exploited (consciously or otherwise) when, in my role of problem-solver, I chose to apply the rules. Although this knowledge is missing in FOO, it was certainly used in operationalization, and is of interest for that reason; an operationalization problem-solver would presumably need such knowledge. Also, the missing rule conditions are interesting as a *potential locus for knowledge acquisition*. That is, they suggest useful kinds of things to learn about a domain, whether from experience or by asking an expert (§8.1.7).

This section tries to characterize domain properties corresponding to such missing rule conditions, although some of them are not readily expressible in domain-related terms. (§6.6.1) discusses knowledge about the capabilities of the runtime mechanism: the conditions it can affect (§6.6.1.1), the information it can know (§6.6.1.2), and the processes occurring in the task environment (§6.6.1.3). (§6.6.2) discusses knowledge about domain concepts with certain mathematical properties, such as monotonic and injective functions. (§6.6.3) discusses rules that exploit the assumption that distinct constants denote distinct entities. (§6.6.4) discusses the validity of rules for manipulating logical connectives.

6.6.1. Knowledge about Runtime Limitations

Some rule conditions missing in FOO involve knowledge about what is and isn't possible in the task environment: what actions can be performed, what quantities can be evaluated.

6.6.1.1 Immutable Conditions

A condition that can't be changed by any action in the task domain is called *immutable, e.g.*:

```
(IN C3 (DIFF (CARDS) (SET QS)))
(IN-SUIT C1 S0)
(IN C1 (CARDS))
```

These conditions are *immutable* in that no legal action in Hearts can affect them: the set of cards is constant, the suit of a card can't be changed, and a card can't be changed into the Queen of spades nor can the Queen be turned into another card (however much some players might wish).

The following rule is logically valid (preserves equivalence) when $P_1 \ldots P_n$ are immutable:

RULE35: (affect (and P_1 ... P ... P_n)) -> (and [affect P] P_1 ... P_n)

Knowledge about what conditions are inherently immutable in a given task domain seems useful in general. One way to deduce it automatically would assume that the knowledge base contains all actions possible (or legal) in the task environment. A condition neither caused nor undone by any known action would be considered immutable.

6.6.1.2 Evaluability

A similar kind of knowledge involves the ability to predict what will be *evaluable* at runtime. This point is illustrated in DERIV9 by the use of the disjoint subsets method (§2.3):

RULE189: (P ... S ...) -> (h (disjoint S S')) for suitable h and S',
[where S will be unknown at runtime]

As implemented, FOO does not test the bracketed condition. The problem in DERIV9 is to find a way of deciding whether an opponent P0 is void in suit S0:

```
(EVAL (EXISTS C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0)))
```

RULE189 is relevant to this problem because player ME knows the size of an opponent's hand but not its contents. If the opponent's hand were visible, it would not be necessary to use the disjoint subsets method; if the size of the hand were unknown, it would not be possible to use themethod. Thus the decision to use the method is based on knowledge about *what information will be available* to the task agent at runtime (§8.1.1).

Similar considerations figure later in DERIV9. For example, the disjoint subsets method requires finding a set of known size to play the role of S' in RULE189. An obvious candidate is the set `(SET-OF C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0))`. However, this set is unsuitable because player `ME` cannot generally determine its size; indeed, the problem of doing so subsumes the problem of telling whether player `P0` is void. The set `(CARDS-IN-SUIT S0)` always has 13 elements and is therefore a suitable S'. The method also requires a common superset U of S and S' whose size is known. The most obvious common superset of S and S' is their union, but its size is unknown for this problem; instead, `(CARDS)` is used, since its size is known to be 52. Thus the choice of S' and U is based on knowledge about which set sizes can be determined at runtime.

Later the solution is refined by finding a predicate P satisfied by all of S, and using it to filter S' and U. To evaluate the refined expression, player `ME` must be able to compute the sizes of (filter S' P) and (filter U P). The predicate `OUT` is used for P, since player `ME` can compute the total number of cards out and (ignoring the hole card) the number of cards out in suit `S0`. Subsequently, functional dependence analysis (§2.8.2) is used to produce a 0th order approximate solution, and the quantity `(#CARDS-OUT-IN-SUIT S0)` is chosen as the independent variable because it's evaluable and its possible values are ordered (otherwise functional dependence wouldn't apply). In short, applying the disjoint subsets method and functional dependence analysis to this problem requires *knowledge about what can be evaluated at runtime using the data available to the task agent.*

Knowledge about what can and can't be evaluated at runtime could be discovered by trial and error. An *analytic* approach would decide the evaluability of a given expression by trying to transform it into an operational expression. An *empirical* approach would record how often the expression was successfully evaluated at runtime (§8.1.7).

6.6.1.3 Processes

Another kind of knowledge about actions in the task domain involves *processes* to which advice refers. For example, these rules assume that the set or sequence S is being extended:

RULE277: (change (f S)) -> (R [f (change S)] [f S]), where f is extremum for the ordering R

RULE300: (next (f S)) -> (f (change S)) if (R [f (change S)] [f S]), where f is extremum for R

Here (change S) denotes (diff (next S) S) but (change (f S)) denotes (≠ [f S] [next (f S)]). The fact that CANTUS-1 is being extended is used in DERIV14 to show that adding new notes to CANTUS-1 changes the climax if the highest new note is higher than the highest old note:

```
(CHANGE (HIGHEST CANTUS-1))
14:42 --- [REDUCE by RULE277] --->
(HIGHER [HIGHEST (CHANGE CANTUS-1)] [HIGHEST CANTUS-1])
```

The same fact is used to deduce that the next climax is the highest new note if it's higher than the old climax:

```
(NEXT (HIGHEST CANTUS-1))
14:55-57 --- [REDUCE by RULE300, analysis] --->
(HIGHEST (CHANGE (CANTUS-1)))
since (HIGHER [HIGHEST (CHANGE CANTUS-1)] [HIGHEST CANTUS-1])
```

These transformations are invalid if S is contracting, *e.g.*, if CANTUS-1 is being shortened.

6.6.2. Monotonic and Injective Properties of Domain Concepts

Another kind of knowledge involves certain mathematical properties of functions.

6.6.2.1 Monotonicity

The concept of a function *monotonic* with respect to an ordering is illustrated by the conditions for the validity of

RULE30: (R [f S] [f S']) -> T if (subset S S'), where R is transitive

RULE30 assumes that (subset S S') implies (R [f S] [f S']), *i.e.*, that f is some kind of *measure* relative to the ordering R and is *monotonic* with respect to the subset relation (the measure of a set is an upper bound on the measure of any of its subsets). The use of RULE30 in DERIV13 depends on knowing this monotonicity relationship for R = SUBSET and f = INTERVALS-OF, *i.e.*, that a subsequence of a cantus contains a subset of its intervals:

```
(SUBSET [INTERVALS-OF PROJECT1] [INTERVALS-OF ...])
13:23 --- [REDUCE by RULE30] --->
(SUBSET PROJECT1 ...)
```

(Note that SUBSET is used ambiguously to mean both "subset" and "subsequence.") The monotonicity relationship also holds for R = =< and f = RANGE, *i.e.*, the range of a subsequence is at most the range of the cantus:

```
(=< [RANGE PROJECT1] [RANGE ...])
13:46 --- [REDUCE by RULE30] --->
(SUBSET PROJECT1 ...)
```

Monotonicity holds for R = `=<` and f = `#`, *i.e.*, a set is at least as big as any subset of it:

```
(=< [# (SET-OF X3 PROJECT1 (= X3 CLIMAX1))]
    [# (SET-OF X4 ... (= X4 CLIMAX1))])
13:96 --- [REDUCE by RULE30] --->
(SUBSET (SET-OF X3 PROJECT1 (= X3 CLIMAX1))
        (SET-OF X4 .... (= X4 CLIMAX1)))
```

6.6.2.2 Injectiveness

The concept of an *injective* function is important in analyzing the validity of

RULE170: (project f (diff S S')) -> (diff [project f S] [project f S'])

This rule preserves semantic equivalence provided the condition f(S-S') = f(S) - f(S') holds. For example, consider the transformation

```
(PROJECT HAND (DIFF (PLAYERS) (SET ME)))
4:12 --- [DISTRIBUTE by RULE170] --->
(DIFF [PROJECT HAND (PLAYERS)] [PROJECT HAND (SET ME)])
```

The validity of this transformation depends on the fact that `HAND` is *injective* (one-to-one), *i.e.*, each player has a *distinct* hand. The condition that f be injective is a sufficient condition for preserving equivalence, but is not strictly necessary. RULE170 preserves equivalence except when S-S' contains an x such that f(x) is in f(S'). In other words, step `4:12` would still be valid even if the rules of Hearts allowed players to share the same hand, provided no other player in `(PLAYERS)` shared `(HAND ME)`.

Another rule guaranteed to be valid for injective f is

RULE172: (in (f e) (project f S)) -> (in e S)

```
(IN (HAND ME) (PROJECT HAND (PLAYERS)))
4:23 --- [REDUCE by RULE172] --->
(IN ME (PLAYERS))

(IN (CARD-OF ME) (PROJECT CARD-OF (PLAYERS)))
5:23 --- [REDUCE by RULE172] --->
(IN ME (PLAYERS))
```

The latter example is based on the fact that `CARD-OF` is injective, *i.e.*, two players can't play the same card in the same trick, or for that matter in the same round. (The `CARD-OF` function implicitly depends on the current trick; it was unnecessary for the purposes of the derivations to bother making this an explicit argument, since none of them refer to cards played in separate tricks.) RULE172 is valid except when there exists an x not in S such that f(x) is in f(S). That is, steps 4:23 and 5:23 are valid even if HAND and CARD-OF are not injective, so long as nobody in (PLAYERS) shares a hand with or plays the same card as a player outside the game.

The point of these rather detailed examples is that monotonicity and injectivity are useful in describing sufficient conditions for a rule to be valid, but necessary conditions for validity are hard to pinpoint.

6.6.2.3 Equivalence-preserving Partial Matching

The injective property is closely related to the validity of partial matching as an equivalence-preserving transformation. For example, consider

RULE43: (R [f e_1 ... e_n] [f e_1' ... e_n']) -> (and [= e_1 e_1'] ... [= e_n e_n']) if R is reflexive

The following transformation is logically valid (preserves equivalence) because HAND is injective:

```
(= [HAND P1] [HAND (QSO)])
3:55 --- [REDUCE by RULE43, simplification] --->
(= P1 (QSO))
```

A general (sufficient) condition for RULE43 to be valid is that

f is *injective* and the relation R *coincides with the equality relation on the range of f*.[11]

The second condition follows from the requirement that the rule's left side imply its right side:

(R [f x_1 ... x_n] [f y_1 ... y_n]) => $x_1 = y_1 \wedge ... \wedge x_n = y_n$ [l.h.s. => r.h.s.]

By the substitution property of equality:

$x_1 = y_1 \wedge ... \wedge x_n = y_n$ => (f x_1 ... x_n) = (f y_1 ... y_n)

Thus

(R [f x_1 ... x_n] [f y_1 ... y_n]) => (f x_1 ... x_n) = (f y_1 ... y_n)

I.e., R restricted to the range of f must be a subset of the equality relation if RULE43 is to be valid for all possible values of x_1 ... x_n and y_1 ... y_n. Since R is reflexive by hypothesis, R restricted to the range of f must coincide with equality. This means in turn that f is injective, *i.e.*,

[11] I owe this to Jim Saxe.

$(f\,x_1 \ldots x_n) = (f\,y_1 \ldots y_n]) \Rightarrow x_1 = y_1 \wedge \ldots \wedge x_n = y_n$

This is easily proved:

$(f\,x_1 \ldots x_n) = (f\,y_1 \ldots y_n]) \Rightarrow (R\,[f\,x_1 \ldots x_n]\,[f\,y_1 \ldots y_n])$ [R is reflexive]
$(R\,[f\,x_1 \ldots x_n]\,[f\,y_1 \ldots y_n]) \Rightarrow x_1 = y_1 \wedge \ldots \wedge x_n = y_n$ [l.h.s. => r.h.s.]

The condition formulated above is satisfied for R = =>, f = MOVE:

```
(=> [MOVE C4 (HAND P1) POT] [MOVE C3 (HAND (QSO)) LOC3])
3:52 --- [REDUCE by RULE43] --->
(AND [= C4 C3] [= (HAND P1) (HAND (QSO))] [= POT LOC3])
```

The function MOVE is injective because movements are distinct events if their objects, origins, or destinations differ. In fact, every action concept in the knowledge base is injective for similar reasons. The implication relation => coincides with equality on the range of MOVE, *i.e.*, the set of movement events, because only one movement can occur at a time according to the model of the Hearts environment represented in the knowledge base.

The same condition is satisfied for R = DURING, f = TAKE:

```
(DURING [TAKE (TRICK-WINNER) C1] [TAKE ME QS])
5:12 --- [REDUCE by RULE43] --->
(AND [= (TRICK-WINNER) ME] [= C1 QS])
```

The function TAKE is injective because it's an action, and DURING coincides with equality on the range of TAKE, *i.e.*, the set of card-taking events, because two cards can't be taken simultaneously according to the simplistic model of the domain.

The general validity condition is *violated* for R = =>, f = HAS and for R = =>, f = VOID:

```
(=> [HAS P0 X2] [HAS P1 X2])
9:33 --- [REDUCE by RULE43] --->
(AND [= P0 P1] [= X2 X2])

(=> [VOID P1 (SUIT-LED)] [VOID P0 S0])
11:11 --- [REDUCE by RULE43] --->
(AND [= P1 P0] [= (SUIT-LED) S0])
```

The relation => doesn't coincide with = on the set of truth values, since NIL => T but NIL ≠ T. Nor are HAS and VOID injective: they can map different combinations of arguments to the same truth value. Since both are fluents, the test for injectiveness is somewhat weaker: a fluent is injective if it doesn't map different combinations of arguments to the same value *at the same point in time*. HAS and VOID fail even this weaker property. In fact, in both examples one can describe situations in which the first expression is true and the second is false. For instance, if player P0 doesn't have card X2, then (=> [HAS P0 X2] [HAS P1 X2]) is vacuously true for any P1. The situation needn't involve vacuously true implication; one can imagine situations where player P1's being void in (SUIT-LED) implies that another player P0 is void in S0 (*e.g.*, if one card in (SUIT-LED) and another in suit S0 are the only cards still out after the last trick of a three-player game is led).

In short, not only is the general (sufficient) condition violated in each case, but the transformation actually produces an expression that means something different from the original one. Nonetheless, both 9:33 and 11:11 make heuristic sense: they contribute to the construction of operational solutions. One explanation of this apparent paradox is that the connective => actually denotes some kind of *relevant implication* stronger than predicate calculus implication (§5.9.2.2.1). Another explanation is that the transformations are *heuristically valid* because they preserve semantic equivalence in all but a few perverse situations (§5.3.3).

6.6.2.4 Soundness

However, a much simpler explanation is that in both cases the transformation is part of an effort to find a *sufficient condition* for a given expression. In DERIV9, the problem is to prove that all of (CARDS-IN-HAND P0) are OUT; in DERIV11, to find a sufficient condition for (VOID P0 S0). Since the right side of RULE43 implies the left side, it is guaranteed to be logically *sound*, and therefore semantically appropriate in constructing a proof or in finding a sufficient condition. A transformation is *sound* if it never transforms falsehood into truth. Thus a sequence of sound transformations that reduce a proposition to T constitutes a proof of it.

6.6.2.5 Problem-solving Goal as an Implicit Rule Condition

This example shows how the applicability of a transformation can depend not only on the expression to be transformed, but also on the *purpose* of the transformation relative to the current problem-solving goal. If the goal is to reduce a problem to a simpler but *equivalent* one, the transformation must be *valid*; if the goal is to reduce a proposition to a simpler *sufficient* condition or to *prove* it (*i.e.*, reduce it to T), the transformation need only be *sound*.

6.6.3. Knowledge about Distinct Entities

A basic assumption of FOO's knowledge representation is that

Distinct constants denote distinct entities.

In DERIV3, this assumption reflects the fact that spades and hearts are distinct suits:

```
(NEQ S H)
3:31 --- [COMPUTE by RULE236] --->
T
```

The idea of a *collection of distinct objects* is important in

RULE288: (# (collection e_1 ... e_n)) -> n

RULE288 is only valid when e_1 ... e_n represent distinct entities. The distinctness property figures as an invariant in

RULE171: (project f (set e_1 ... e_n)) -> (set (f e_1) ... (f e_n))

RULE171 preserves the distinctness property, *i.e.*, produces a list of expressions denoting distinct entities *when e_1 ... e_n is such a list, unless (f e_i) = (f e_j) for some e_i and e_j in e_1 ... e_n where $i \neq j$.* A sufficient condition for RULE171 to preserve distinctness is that f be injective.

6.6.4. Knowledge about Logical Connectives

Several of FOO's rules transform expressions by moving logical connectives such as quantifiers or implication. It appears difficult to express general conditions for the validity of such transformations in terms of easily recognizable properties of a task domain, *i.e.*, to characterize the class of expressions for which they preserve semantic equivalence. The apparent reason for this difficulty is that the transformations are expressed in *structural* terms (§5.10), while validity is a *semantic* property.

For example, consider the function transposition rule (§5.10.1)

RULE58: (during [each x S E_x] A) -> (exists x S (during E_x A))

RULE58 produces a valid (equivalence-preserving) transformation in DERIV5:

```
(DURING [EACH C1 (CARDS-PLAYED) (TAKE (TRICK-WINNER) C1)]
        [TAKE ME QS])
5:11 --- [by RULE58] --->
(EXISTS C1 (CARDS-PLAYED)
   (DURING [TAKE (TRICK-WINNER) C1] [TAKE ME QS]))
```

This transformation is valid since the event `(TAKE ME QS)` can't span more than one event `(TAKE (TRICK-WINNER) C1)`. However, RULE58 can also produce the transformation

```
(DURING [EACH C1 (CARDS-PLAYED) (TAKE (TRICK-WINNER) C1)]
        [EACH C1 (CARDS-PLAYED) (TAKE (TRICK-WINNER) C1)])
--- [by RULE58] --->
(EXISTS C1 (CARDS-PLAYED)
   (DURING [TAKE (TRICK-WINNER) C1]
           [EACH C1 (CARDS-PLAYED) (TAKE (TRICK-WINNER) C1)]))
```

This transformation is obviously invalid, since it transforms truth to falsehood.

It is not clear how to characterize concisely the class of expressions for which RULE58 preserves equivalence. Since the right side implies the left, the validity condition reduces to the requirement (during [each x S E_x] A) => (exists x S (during E_x A)). One can formulate sufficient conditions for validity: RULE58 preserves equivalence if events of type A can't span more than one event of type E, and can always be used in derivations of the form P_1 => ... => P_n. However, there appears to be no obvious *easy-to-test* necessary and sufficient condition for the validity of RULE58.

This problem is even worse for other rules that move quantifiers and implication connectives:

RULE192: (= [Q x S P_x] [Q y S R_y]) -> (= P_y R_y), where y is implicitly universally quantified

RULE220: (=> C [Q x S P_x]) -> (Q x S [=> C P_x])

RULE224: (=> [and C_1 ... C ... C_n] [P ...]) -> (=> C [P ...]) if P occurs in C

RULE225: (=> [or ... C ...] [P ...]) -> (and [not (or ...)] [=> C (P ...)]) if P occurs in C

RULE226: (=> [=> R C] [P ...]) -> (and R [=> C (P ...)]) if P occurs in C

RULE321: (=> [P ...] [and C_1 ... C ... C_n]) -> (and [=> (P ...) C] C_1 ... C_n) if P occurs in C

RULE322: (=> [P e_1 ... e_n] [exists x S (P e_1' ... e_n')]) -> (and [in x S] [= e_1 e_1'] ... [= e_n e_n']),
where x is implicitly existentially quantified -- *combines RULE220 and partial matching*

RULE329: (P ... [Q x S R_x] ...) -> (Q x S [P ... R_x ...]) -- *subsumes RULE220*

Some of these (*e.g.*, RULE192) are logically sound, but the logical status of the others is unclear. For example, if P quantifies over a second variable y appearing in R_x, RULE329 can transform the assertion that every integer is less than some other into the claim that a greatest integer exists:

```
(FORALL Y (INTEGERS) (EXISTS X (INTEGERS) (< Y X)))
--- [by RULE329] --->
(EXISTS X (INTEGERS) (FORALL Y (INTEGERS) (< Y X)))
```

RULE329 transforms truth into falsehood and *vice versa* by transposing NOT with a quantifier:

```
(NOT (FORALL X (INTEGERS) (= X 0)))
--- [by RULE329] --->
(FORALL X (INTEGERS) (NOT (= X 0)))

(NOT (EXISTS X (INTEGERS) (= X 0)))
--- [by RULE329] --->
(EXISTS X (INTEGERS) (NOT (= X 0)))
```

In general, the semantic effect of restructuring an implication or quantified statement depends on the individual expression, and there is no apparent *easily tested* criterion that characterizes the class of expressions for which such a transformation preserves semantic equivalence.

A heuristic approach to this problem is to use such transformations freely and see if they produce effective solutions. If an expectation raised by the solution is subsequently violated in an actual task situation, and the error is traced back to the transformation, the solution can be revised or replaced. The paradigm of learning based on *analysis of violated expectations* is discussed in [Hayes-Roth 81b].

6.7. Applications of Conceptual Knowledge: Summary

Some of the conceptual knowledge used in the derivations is *explicitly encoded* in FOO as definitions and semantic relations; some is *simulated* by rules or assumptions; and some knowledge about when transformations are semantically appropriate is *missing*. Thus conceptual knowledge used in operationalization forms a taxonomy based on how (and whether) it's encoded in FOO:

1. Encoded knowledge (§1.3)
    - a. Explicitly encoded knowledge (§1.3)
        - i. Concept definitions (§1.3.1)
        - ii. Semantic relations (§1.3.2)
            1. Is-a hierarchy (§1.3.2.1)
            2. Attribute-value pairs (§1.3.2.2)
    - b. Knowledge base as implicit model of domain (§6.4) -- definitions as assertions (§6.4.3)
        - i. Well-definedness assumption (§6.4.1)

i. Embedded assertions (§6.4.2)

ii. Distinct-constant *assumption* (§6.6.3)

2. Simulated knowledge

   a. Fact rules (§1.3.3)

      i. Types (§1.3.3.1)

      ii. Partitions (§1.3.3.2)

      iii. Features (§1.3.3.3)

      iv. Deductions (§1.3.3.4)

   b. Explicit assumptions made in derivations (§6.4.4)

3. Missing knowledge corresponding to implicit rule conditions (§6.6)

   a. Runtime capabilities (§6.6.1)

      i. Immutable conditions (§6.6.1.1)

      ii. Processes (§6.6.1.3)

      iii. Evaluability (§6.6.1.2)

   b. Mathematical properties of and relations between concepts (§6.6.2)

      i. Monotonicity (§6.6.2.1)

      ii. Injectiveness (§6.6.2.2)

   c. Rule properties relative to operationalization goals (§6.6.2.5)

      i. Preserve semantic equivalence (§6.6.2.3)

      ii. Transform to sufficient condition (soundness) (§6.6.2.4)

      iii. Preserve distinctness of enumeration (§6.6.3)

   d. Effect of moving logical connectives (§6.6.4)

The conceptual knowledge encoded in FOO is used in various ways (§6). An obvious one, and in fact the most common, is to plug in the definition of a concept (§6.2). However, the more interesting applications of conceptual knowledge are those that achieve useful results without expanding the problem down to the lowest level of detail. For instance, long chains of reasoning can be skipped by exploiting stored semantic relations between concepts (§1.3.2), such as homomorphic mappings (§6.1.2). This is facilitated by a rule matcher that applies general rules to (domain-) specific problems by searching through an is-a hierarchy (§6.1). Other methods search the knowledge base of concept definitions for concepts of a particular form (§6.5). This creates the potential for extending the range of derivable operationalizations simply by defining a new concept (§6.5.5).

Applications of conceptual knowledge in FOO fit into a taxonomy:

1. Translation (§5.8)
    a. Elaboration -- expand definition (§6.2)
        i. Map problem to general method (§5.8.2)
        ii. Split problem into subproblems (§5.7.4)
    b. Recognition -- identify known concept by matching (§6.3) or analysis (§6.3.2)
        i. Find action with specified effect (§2.4)
        ii. Find learned methods indexed under concept (§6.3.1)
2. Search through concept definitions (§6.5)
    a. Intersection search based on closure assumption (§6.5.1)
    b. Plausible generation based on useful-concept assumption (§6.5.2)
        i. Find sufficient condition (§6.5.2)
        ii. Find action with specified effect (§6.5.3)
        iii. Find necessary condition (§6.5.4)
3. Use semantic relations to connect general and specific concepts (§6.1)
    a. Exploit special cases (§6.1.1)
    b. Apply homomorphisms (§6.1.2)
    c. Delete identity elements (§6.1.3)
    d. Translate between related adjectives (§6.1.4)

Most of FOO's domain knowledge is encoded as LISP function definitions (§1.3.1). Different parts of speech correspond to functions of different forms (§1.3.1.3). Since FOO deals only with well-formed input, its knowledge representation lacks features like slot restrictions (§1.3.1.1) designed to facilitate processes of disambiguation, instantiation, and completion that deal with the informality of natural language (§8.2.1). Such features were simulated by rules when needed (§1.3.3), notably in instantiating missing components of methods (§2.3.1.4) (§3.3).

Chapter 7
Rules as Operators for Means-End Analysis

As we have seen, analysis is the glue that holds the operationalization process together: it makes the connection between a problem stated in one set of terms and a method stated in another. Consider the problem of automatically choosing which rule to apply at each point, at least for making these sorts of connections and solving the subproblems generated by the higher-level operationalization methods. An obvious approach would use *means-end analysis* ("MEA") [Newell 69] to guide rule selection. What knowledge about the rules is required to use them as operators in a problem reduction mode for translating expressions to a desired form? For example, can the rules be characterized as translating between abstract *regions* of expressions corresponding to different languages, *e.g.*, numbers, sets, quantifiers? If so, could a rule sequence for translating an expression from one region to another be found by an intersection search through an abstracted network with nodes for the regions and arcs for the rules?

(§7.3) models the main proof in DERIV12 as the product of a MEA problem-solver in order to explore the sorts of differences and goal descriptions that might be needed. As an aid to understanding this extended example, (§7.1) defines three kinds of differences between expressions, and (§7.2) illustrates them by means of a simple example. (§7.4) summarizes the lessons drawn from the examples.

7.1. Differences between Expressions

To get an idea of how FOO's rules might be used as operators, I tried to analyze what the GPS-type differences [Newell 60] would be in a typical analysis. The differences in the example include *mismatches* between current and desired expression (§7.1.1), *disagreement* between two sub-expressions which must agree in order to match a rule condition (§7.1.2), and *structural* differences between a current expression and a desired expression with the same symbols arranged differently (§7.1.3).

7.1.1. Corresponding Symbols Mismatch

The simplest kind of difference between the current expression (f ...) and a desired expression (g ...) is when f ≠ g, or more precisely, the condition f is-a g does not hold. A rule sequence for eliminating this difference may be found by searching a network whose nodes are the top-most symbols of the rule left- and right-hand sides, and whose arcs correspond to the rules in the obvious way. Thus a rule (f' ...) -> (g' ...) would be represented in the network by the arc f' -> g'. The search algorithm must also allow any number of is-a links. For instance, if f is-a f' and g is-a g', the search would retrieve the rule (f' ...) -> (g' ...) as a candidate for reducing the difference between (f ...) and (g ...).

The network representation for a rule of the form (f x_1 ... x_k) -> (f y_1 ... y_k) is more complicated, since such a rule doesn't change the top-most symbol of the expression to which it's applied. Such a rule is represented as *several* arcs x_i -> y_i. If x_i ≠ y_i, arcs between the corresponding unequal components of x_i and y_i are included, and so forth. These arcs represent *context-sensitive* transformations, *i.e.*, transformations x_i -> y_i that only apply when x_i occurs inside an expression of the form (f x_1 ... x_k). An example of a rule that leaves the top-most symbol unchanged is

RULE161: (Q x S (P ... (f x) ...)) -> (Q y (project f S) (P ... y ...))

RULE161 transforms a quantified expression into another with the same top-most symbol Q. It would be represented in the network by two arcs. The arc `SET -> PROJECT` represents the difference between S and (project f S), and the arc `FUNCTION -> VARIABLE` represents the difference between (f x) and y.

The elaboration rule RULE124 and the recognition rule RULE123 would have to be treated specially to avoid short-circuiting the search mechanism. For example, representing RULE123 as `ANY -> DEFINED` would cause it to be suggested as a one-arc solution to almost any search problem. To avoid this problem, RULE123 could be represented as a *set* of arcs f -> g for each concept f whose definition is of the form (lambda (x_1 ... x_n) (g ...)). For example, these arcs would include `MOVE -> GET-CARD`, `MOVE -> PLAY`, and `MOVE -> TAKE` corresponding to the definitions

```
GET-CARD = (LAMBDA (P C) (MOVE C ?LOC (HAND P)))
PLAY = (LAMBDA (P C) (MOVE C (HAND P) POT))
TAKE = (LAMBDA (P C) (MOVE C POT (PILE P)))
```

RULE124 would be represented by the same arcs going in the opposite direction. This more specific encoding would make it possible to find connections between concepts related by their definitions without finding spurious shortcuts between other concepts.

7.1.2. Internal Disagreement

Another kind of difference is violation of an agreement condition implied by the occurrence of the same variable in more than one place in a rule condition. Consider an expression of the form (=> (f ...) (g ...)), where $f \neq g$. It violates the condition $f = g$ required to satisfy the left-hand side of a matching rule like

RULE43: (R (f x_1 ... x_n) (f y_1 ... y_n)) -> (and (= x_1 y_1) ... (= x_n y_n))

Eliminating a difference of this kind appears to require rules like

To express (f ...) and (g ...) in common terms, expand (f ...) in terms of (g ...).

A search procedure for this rule might search through the knowledge base of concept definitions to see if f were defined directly or indirectly in terms of g. FOO already has a procedure that does exactly this kind of search for other reasons (§5.6).

7.1.3. Structural Differences

Sometimes one can find a path from a *sub-expression* of the current expression to a desired form. Here the difference is one of non-adjacency. Suppose the current expression is (f (g ... e ...)), the desired form is (f e'), and there is a path from e to e'. The problem is that e is not adjacent to f. A solution may be to bring e closer to f by applying a transposition rule (f (g ...)) -> (g (f ...)), *e.g.*,

RULE329: (P ... (Q x S Ex) ...) -> (Q x S (P ... Ex ...))

RULE329 moves quantifier Q outside, thereby lifting Ex closer to P. This may eventually make it possible to apply a rule whose left-hand side has the desired form (P E). In general, structural differences can be eliminated by *restructuring* rules (§5.10).

7.2. A Simple Example of Means-End Analysis

The differences and the techniques for reducing them discussed in (§7.1) can be illustrated by showing how they could be used to translate into the language of the pigeon-hole principle the problem of deciding whether the Queen of spades is out (§5.8.1). The translation begins by successive elaboration:

```
(QS-OUT)
4:1 --- [ELABORATE by RULE124] --->
(OUT QS)

4:2 --- [ELABORATE by RULE124] --->
(EXISTS P1 (OPPONENTS ME) (HAS P1 QS))

                                   (HAS P1 QS)
4:3                                --- [ELABORATE by RULE124] --->
                                   (AT QS (HAND P1))

4:4                                --- [ELABORATE by RULE124] --->
                                   (= (LOC QS) (HAND P1))
```

This is followed by a change of variable using

RULE161: (Q x S [P ... (f x) ...]) -> (Q y (project f S) [P ... y ...])

The effect is to *restructure* the quantification (§5.10):

```
(EXISTS P1 (OPPONENTS ME) [= (LOC QS) (HAND P1)])
4:5 --- [transfer HAND by RULE161] --->
(EXISTS Y1 (PROJECT HAND (OPPONENTS ME)) [= (LOC QS) Y1])
```

The equality, quantified over hands instead of players, is recognized in terms of set membership:

```
4:6 --- [RECOGNIZE by RULE162] --->
(IN (LOC QS) (PROJECT HAND (OPPONENTS ME)))
```

The recognition is performed by

RULE162: (exists x S (= e x)) -> (in e S)

The resulting expression matches the problem statement for the pigeon-hole principle (§2.2):

RULE169: (in (f ...) S) -> (not (in (f ...) (diff (Range f) S)))

Suppose the initial goal is to reformulate (QS-OUT) in the form (IN ...) so that the pigeon-hole principle can be applied. (The interesting problem of how this goal might be adopted in the first place is outside the scope of this discussion (§4.2.2.1) (§8.1.1).) How might the transformation sequence 4:1-6 be generated automatically?

First, the mismatch between the top-level symbols QS-OUT and IN suggests searching the network representation of the rules for a path between them (§7.1.1). Such a path exists:

```
QS-OUT --RULE124--> OUT --RULE124--> EXISTS --RULE162--> IN
```

This path constitutes a top-level plan for translating (QS-OUT) into the desired form. The first two steps of this plan are immediately executable, and correspond to 4:1-2.

Now a snag develops: the expression (EXISTS P1 (OPPONENTS ME) (HAS P1 QS)) does not match the left-hand side of the next rule in the plan, namely

RULE162: (exists x S (= e x)) -> (in e S)

As before, the mismatch between the corresponding symbols HAS and = suggests finding a path between them. Such a path exists:

```
HAS --RULE124--> AT --RULE124--> =
```

Steps 4:3-4 correspond to the rule applications indicated by this path.

The resulting expression (EXISTS P1 (OPPONENTS ME) (= (LOC QS) (HAND P1))) still does not match the left-hand side of RULE162, but the mismatch is now of a different nature: internal disagreement (§7.1.2) between the quantifier variable P1 and the second argument (HAND P1) of the quantified equality, both of which correspond to x in RULE162. However, this can be viewed as a structural difference, since the desired argument P1 occurs as a sub-expression of the actual argument (HAND P1) (§7.1.3). One way to move the function HAND out of the equality is to *transfer* it (§5.10.2) to another component of the expression containing it. This is done at step 4:5 by a restructuring rule:

RULE161: (Q x S [P ... (f x) ...]) -> (Q y (project f S) [P ... y ...])

RULE161 transfers f from a quantified condition to the component describing the scope of quantification. Here f = HAND.

This produces (EXISTS P1 (PROJECT HAND (OPPONENTS ME)) [= (LOC QS) Y1]), enabling the top-level plan to be completed by applying RULE162 at step 4:6.

The resulting expression matches the left-hand side of RULE169, so the initial goal of translating (QS-OUT) into the language of the pigeon-hole rule has been achieved. The point of this example was to show how a sequence of rules applied in DERIV4 could have been chosen automatically using means-end analysis. The hierarchical plan underlying this sequence is summarized below. The underlined prerequisites of steps in the plan are subgoals to be satisfied at a lower level of detail. The top-level goal is to apply RULE169 to the expression (QS-OUT).

```
RULE169: IN (f x) S -> apply pigeon-hole principle
4:1 RULE124: QS-OUT -> OUT
4:2 RULE124: OUT -> EXISTS
4:6 RULE162: EXISTS x S (= e x) -> IN e S
4:3     RULE124: HAS -> AT
4:4     RULE124: AT -> =
4:5     RULE161: Q x S (= e (f x)) -> transfer f to S
```

7.3. An Extended Example of Means-End Analysis

This section presents a MEA model of the main proof in DERIV12 (§2.4.2), *i.e.*, tries to explain how the sequence of rules used could have been determined by means-end analysis. In the process, it makes several observations about the kinds of information a means-end analysis component would need to know about the rules, and how a substantial portion of it could be encoded in a network and retrieved by intersection search.

At each step in the example, the simulated problem-solver has a *current expression* and a *desired expression*, either of which may be a somewhat abstracted description of a class of expressions. The difference between the two is used to suggest a rule, sequence of rules, or abstract operation (*e.g.*, lifting a sub-expression) to reduce or eliminate the difference. If a proposed rule is not immediately applicable, the problem-solver establishes a subgoal whose desired expression is the left-hand side of the rule. Once the subgoal is achieved, the rule is applied. The example derivation exhibits a rather nested structure; the first rule suggested is not applied until the last step, and the basic plan (rule sequence) for most of the derivation is created before the first step is actually executed. The note-difference/find-rule/apply-rule cycle is shown in the notation:

```
                / <current expression>
                \ <desired expression>

                  Find <rule>:  <lhs> -> <rhs>
                  <unsatisfied lhs or rule condition>
                       <subgoal ...>

                  <current expression>
d:n               --- [<tranformation type> by <rule>] --->
                  <new expression>
```

The assumed goal at the beginning of the example is to apply

RULE234: P -> (was-during (current A) P) if (not (during (A) (undo P)))

(As in (§7.2), the origin of this goal is beyond the concern of the example.) Here P = (VOID P0 S0), A = ROUND-IN-PROGRESS. Thus the initial expression is

```
                    (NOT (DURING (ROUND-IN-PROGRESS) (UNDO (VOID P0 S0))))
```

The goal is to verify this condition, *i.e.*, transform it to T. This is represented by the difference

```
                  / (NOT (DURING (ROUND-IN-PROGRESS) (UNDO (VOID P0 S0))))
                  \ T
```

The initial expression is a negated condition, while the desired form is the truth constant:

```
                  / NOT
                  \ T
```

Thus the top-level difference between the two is a mismatch between the corresponding symbols NOT and T (§7.1.1). An intersection search through the abstract rule network described earlier would find the arc COMPUTABLE -> CONSTANT corresponding to

RULE236: $(f c_1 \ldots c_n) \rightarrow c$,
where c is the result of applying the computable function f to the constant arguments $c_1 \ldots c_n$

RULE236 transforms a computable function of constant arguments into a constant. The search path would include the arcs NOT is-a COMPUTABLE and T is-a CONSTANT:

```
        NOT                      T
        is-a                     is-a
      COMPUTABLE --RULE236--> CONSTANT
```

Thus the simple network representation contains enough information to suggest using RULE236:

```
                        Find RULE236:  COMPUTABLE -> CONSTANT
                        RULE236 condition:  arguments must be constants
```

RULE236 requires that all the arguments be constants. The single argument to NOT in the initial expression is not a constant. This mismatch must be fixed before RULE236 can be applied.

```
                         / (DURING (ROUND-IN-PROGRESS)
                        /          (UNDO (VOID P0 S0)))
                        \  CONSTANT
```

An intersection search between DURING and <constant> would find the path

```
      DURING --RULE57--> NIL --is-a--> CONSTANT
```

The first arc of this path represents

RULE57: (during s e) -> nil if s and e have no sub-events in common

The second arc simply represents the fact that NIL is a constant.

```
                                        Find RULE57:  DURING -> NIL
                                        RULE57 condition: args refer to known actions
```

In order to compute the sub-events of s and e, they must be expressed in terms of known action concepts. ROUND-IN-PROGRESS is such a concept, but UNDO is not. This is a mismatch.

```
                                             / (UNDO (VOID PO SO))
                                             \ (... KNOWN-ACTION ...)
```

An intersection search between UNDO and KNOWN-ACTION would find the following path:

```
                                                                KNOWN-ACTION
                                                                    is-a
UNDO --RULE377--> CAUSE --RULE358--> MOVE --RULE123--> DEFINED
```

The rules represented by this path are:

RULE377: (undo (not P)) -> (cause P)

RULE358: (cause (at obj loc)) -> (move obj loc' loc), where loc' is the previous location of obj

RULE123: e -> (f e_1 ... e_n) if f is defined as (lambda (x_1 ... x_n) e')
and e is the result of substituting e_1 ... e_n for x_1 ... x_n throughout e'

RULE377 and RULE358 are represented by the respective arcs UNDO -> CAUSE and CAUSE -> MOVE, while RULE123 is represented by a set of arcs (§7.1.1). This set includes the arcs MOVE -> GET-CARD, MOVE -> PLAY, and MOVE -> TAKE corresponding to the definitions

```
GET-CARD = (LAMBDA (P C) (MOVE C ?LOC (HAND P)))

PLAY = (LAMBDA (P C) (MOVE C (HAND P) POT))

TAKE = (LAMBDA (P C) (MOVE C POT (PILE P)))
```

Actually, these concepts define three different paths from UNDO to KNOWN-ACTION:

```
UNDO --...--> MOVE --RULE123--> GET-CARD is-a KNOWN-ACTION

UNDO --...--> MOVE --RULE123--> PLAY is-a KNOWN-ACTION

UNDO --...--> MOVE --RULE123--> TAKE is-a KNOWN-ACTION
```

However, all three paths correspond to the same rule sequence:

```
                                            Find RULE377-RULE358-RULE123
```

This rule sequence gives a high-level plan for reformulating the current expression in terms of a known action. Execution of the plan begins by trying to apply the first rule in the sequence:

```
                                            RULE377 lhs:  (UNDO (NOT ...))
```

The current expression (UNDO (VOID ...)) mismatches the left side of

RULE377: (undo (not P)) -> (cause P)

```
                                                 / VOID
                                                 \ NOT
                                                   Find RULE124:  VOID -> NOT
```

An intersection search would find the arc VOID --RULE124--> NOT based on the definition

```
     VOID = (LAMBDA (P S)
                 (NOT (EXISTS C (CARDS-IN-HAND P) (IN-SUIT C S))))
```

The reasoning up to this point has generated a hierarchical partial *plan* for proving

```
                 (NOT (DURING (ROUND-IN-PROGRESS) (UNDO (VOID P0 S0))))
```

Lower-level plan steps satisfy prerequisites (underlined below) for higher-level steps:

```
     Prove (NOT (DURING (ROUND-IN-PROGRESS) (UNDO (VOID P0 S0))))
         RULE236: NOT CONSTANT -> T
             RULE57: DURING KNOWN-ACTION -> NIL
                 RULE377: UNDO NOT -> CAUSE
                   * RULE124: VOID -> NOT
                 RULE358: CAUSE -> MOVE
                 RULE123: MOVE -> KNOWN-ACTION
```

The starred step is immediately executable, and constitutes the first actual step of the proof:

```
     12:2                                      (VOID P0 S0)
                                               --- [ELABORATE by RULE124] --->
                                               (NOT (EXISTS C1
                                                       (CARDS-IN-HAND P0)
                                                       (IN-SUIT C1 S0)))
```

This satisfies the precondition of RULE377, so apply it:

```
                                         (UNDO (NOT (EXISTS C1
                                                       (CARDS-IN-HAND P0)
                                                       (IN-SUIT C1 S0))))
     12:3                                --- [RECOGNIZE by RULE377] --->
                                         (CAUSE (EXISTS C1 (CARDS-IN-HAND P0)
                                                        (IN-SUIT C1 S0)))
```

The next rule in the 3-rule sequence found earlier is

RULE358: (cause (at obj loc)) -> (move obj loc' loc), where loc' is the previous location of obj

The left side of RULE358 doesn't match the current expression:

```
(CAUSE (EXISTS C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0)))
                                        RULE358 lhs:  (CAUSE (AT ?OBJ ?LOC))

                                             / EXISTS
                                             \ AT
```

Searching for a path from EXISTS to AT would lead to a dead end. More context is needed:

```
                                      / (CAUSE (EXISTS C1 (CARDS-IN-HAND P0)
                                     /           (IN-SUIT C1 S0)))
                                      \  (CAUSE (AT ?OBJ ?LOC))
```

Although (EXISTS C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0)) and (AT ?OBJ ?LOC) mismatch, they are related in that the definition of CARDS-IN-HAND indirectly mentions AT:

```
CARDS-IN-HAND
  defined as
(SET-OF C (CARDS) (HAS P C)) --contains--> HAS --RULE124--> AT
```

This connection could be discovered by an intersection search based on the *definitions*

```
CARDS-IN-HAND = (LAMBDA (P) (SET-OF C (CARDS) (HAS P C)))

HAS = (LAMBDA (P C) (AT C (HAND P)))
```

Such a search has a higher branching factor than the kind described earlier, since it propagates from the node for f to *all* the functions used in the definition of f, not just the top-most one. Note that the network for this search includes a node for each concept plus another for its definition.

The semantic connection between CARDS-IN-HAND and AT suggests using function transposition (§5.10.1) to *embed* CAUSE next to (CARDS-IN-HAND P0):

```
                                        Embed CAUSE next to CARDS-IN-HAND
```

One of FOO's function transposition rules is

RULE329: (P ... (Q x S Ex) ...) -> (Q x S (P ... Ex ...)) -- *embed P closer to Ex*

Why choose RULE329 over other transposition rules? The left side of the chosen rule should resemble the current expression, and its *effect* should match the current goal. One approach to finding such rules uses intersection search. Transposition rules are represented as special arcs, *e.g.*,

```
PREDICATE ==RULE329==> QUANTIFIER
```

If the goal is to embed f next to h in an expression of the form (f ... (g ... (h ...))), the search looks for a path from f to g or from g to h. This is because f can be moved next to g by transposing f with g or g with h. For example, the current goal is to embed CAUSE next to CARDS-IN-HAND in the expression

```
(CAUSE (EXISTS C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0)))
```

An intersection search between CAUSE and EXISTS would find the path

```
  CAUSE                    EXISTS
   is-a                     is-a
PREDICATE ==RULE329==> QUANTIFIER
```

This suggests using RULE329 to transpose CAUSE with EXISTS:

```
Find RULE329:  (P (Q E)) -> (Q (P E))
RULE329 lhs:  (P ... (Q x S Ex) ...)
```

RULE329 can be applied to the current expression, but (CARDS-IN-HAND P0) is in the wrong position to achieve the desired effect of moving it next to CAUSE:

```
 / (EXISTS C1 (CARDS-IN-HAND P0)
/      (IN-SUIT C1 S0))
\  (Q x S (CARDS-IN-HAND P0))
```

This suggests *tranferring* (CARDS-IN-HAND P0) from S into Ex in (Q x S Ex):

```
Transfer (CARDS-IN-HAND P0)
```

One of FOO's transfer rules (§5.10.2) (choosing it is an interesting problem!) is

RULE135: (Q x (set-of y S P_y) R_x) -> (Q x S (and P_x R_x)) -- *move P into quantified condition*

RULE135 transfers P from the scope of the quantifier variable x into the condition quantified over x. Here Q = EXISTS, P_y = at', R_x = (IN-SUIT C1 S0).

```
Find RULE135
RULE135 lhs: (Q x (set-of ...)
                  Rx)
```

However, the current expression (EXISTS C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0)) mismatches the left side (Q x (set-of S Py) Rx) of RULE135:

```
/ CARDS-IN-HAND
\ SET-OF
```

This difference is eliminated by RULE124, suggested by the arc

```
CARDS-IN-HAND --RULE124--> SET-OF
```

```
                                            Find RULE124

                                            (CARDS-IN-HAND P0)
12:4                                        --- [by RULE124] --->
                                            (SET-OF C2 (CARDS)
                                               (HAS P0 C2))
```

The partial plan at this point looks like this:

```
Prove (NOT (DURING (ROUND-IN-PROGRESS) (UNDO (VOID P0 S0))))
    RULE236: NOT CONSTANT -> T
        RULE57: DURING KNOWN-ACTION -> NIL
12:3        RULE377: UNDO NOT -> CAUSE
12:2            RULE124: VOID -> NOT
            RULE358: CAUSE AT -> MOVE
                RULE329: embed CAUSE next to quantified condition
                    RULE135: Q x (SET-OF y S P) R -> transfer P
12:4                    RULE124: CARDS-IN-HAND -> SET-OF
            RULE123: MOVE -> KNOWN-ACTION
```

RULE135 can now be used as suggested to transfer from S to R in (Q x S P). However, the expression transferred is not (CARDS-IN-HAND P0), but (HAS P0 C1). This is alright, because the reason for transferring (CARDS-IN-HAND P0) was the semantic connection between CARDS-IN-HAND and AT, and there exists an even closer connection between HAS and AT. The restructuring goals should not refer to the specific expression (CARDS-IN-HAND P0), but to the intensionally specified "expression defined in terms of AT."

This is an opportune point to note that the example would be much easier to describe if step 12:8, where (HAS P0 C1) is expanded in terms of AT, occurred earlier in the derivation. When I originally generated DERIV12, I deferred this step as long as possible in order to keep the current expression shorter. There are actually several differences between the expression following 12:3 -- (CAUSE (EXISTS C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0))) -- and RULE358's left side (cause (at obj loc)). (Recall that RULE358 is the second rule in the high-level 3-step plan for translating (UNDO (VOID P0 S0)) into a known action.) The argument to CAUSE should be AT. Instead, it *contains* (rather than *is*) an expression *indirectly* (rather than *directly*) *defined in terms of* (rather than *equal to*) the desired symbol AT.

DERIV12 addresses the structural difference before the semantic mismatch: (CARDS-IN-HAND P0) is moved closer to CAUSE before expanding it in terms of AT. It is expanded into (SET-OF C2 (CARDS) (HAS P0 C2)) at step 12:4 only so that RULE135 (and consequently

RULE329) can be applied. An automatic problem-solver might find it easier in general to *resolve semantic mismatches before reducing structural differences.*

RULE135 *transfers* the expression (HAS PO C2) into the desired position:

```
                                   (EXISTS C1 (SET-OF C2 (CARDS)
                                                 (HAS PO C2))
                                      (IN-SUIT C1 SO))
12:5                               --- [transfer by RULE135] --->
                                   (EXISTS C1 (CARDS)
                                      (AND [HAS PO C1]
                                           [IN-SUIT C1 SO]))
```

Now (HAS PO C1) is in the right position for RULE329 to embed CAUSE closer to it:

```
                            (CAUSE (EXISTS C1 (CARDS)
                                      (AND [HAS PO C1]
                                                [IN-SUIT C1 SO])))
12:6                        --- [transpose by RULE329] --->
                            (EXISTS C1 (CARDS)
                               (CAUSE (AND [HAS PO C1]
                                           [IN-SUIT C1 SO])))
```

However, HAS is still not adjacent to CAUSE as it must be to match RULE358's left side:

```
/ (CAUSE (AND (HAS ...) ...)
\ (CAUSE (AT ?OBJ ?LOC))
```

The goal at this point is to embed CAUSE next to HAS in the expression

```
(CAUSE (AND [HAS PO C1] [IN-SUIT C1 SO]))
```

An intersection search between CAUSE and AND (§7.3) would find the path

```
CAUSE is-a AFFECT ==RULE35==> AND
```

This suggests that CAUSE can be embedded next to HAS by the transposition rule

RULE35: (C (and P_1 ... P_n)) -> (and [C P_i] P_1 ... P_{i-1} P_{i+1} ... P_n),
where C denotes change over time, *e.g.*, cause or undo -- *embed C next to P_i*

Here C = CAUSE, P_i = (HAS PO C1):

```
Embed CAUSE next to HAS
Find RULE35
```

RULE35 is directly applicable to the current expression, so apply it:

```
                              (CAUSE (AND [HAS P0 C1]
                                          [IN-SUIT C1 S0]))
12:7                          --- [REDUCE by RULE35] --->
                              (AND [CAUSE (HAS P0 C1)]
                                   [IN-SUIT C1 S0])
```

This brings HAS next to CAUSE, but the resulting expression (CAUSE (HAS P0 C1)) does not yet match the left side of the second rule in the 3-step plan:

RULE358: (cause (at obj loc)) -> (move obj loc' loc), where loc' is the previous location of obj

```
                                          / HAS
                                          \ AT
```

The arc HAS --RULE124--> AT suggests how to eliminate this difference:

```
                                          Find RULE124

                                          (HAS P0 C1)
12:8                                      --- [RULE124] --->
                                          (AT C1 (HAND P0))
```

RULE358 can at last be applied:

```
                                  (CAUSE (AT C1 (HAND P0)))
12:9                              --- [by RULE358] --->
                                  (MOVE C1 LOC1 (HAND P0))
```

RULE358 was the second rule in a 3-step plan for reformulating (UNDO (VOID P0 S0)) in terms of a known action:

```
           (1)            (2)              (3)
UNDO --RULE377-> CAUSE --RULE358-> MOVE --RULE123-> KNOWN-ACTION
```

When this plan was constructed, it wasn't known which action MOVE would be recognized as, but only GET-CARD matches the current expression:

```
                                  (MOVE C1 LOC1 (HAND P0))
12:10                             --- [by RULE123] -->
                                  (GET-CARD P0 C1 LOC1)
```

The proof plan at this stage of partial execution looks like this:

```
        Prove (NOT (DURING (ROUND-IN-PROGRESS) (UNDO (VOID PO SO))))
            RULE236: NOT CONSTANT -> T
                RULE57: DURING KNOWN-ACTION -> NIL
12:3            (1) RULE377: UNDO NOT -> CAUSE
12:2                    RULE124: VOID -> NOT
12:9            (2) RULE358: CAUSE AT -> MOVE
12:6                    RULE329: embed CAUSE next to quantified condition
12:5                        RULE135: Q x (SET-OF y S P) R -> transfer P
12:7                    RULE35: embed CAUSE next to HAS
12:8                    RULE124: HAS -> AT
12:4                            RULE124: CARDS-IN-HAND -> SET-OF
12:10           (3) RULE123: MOVE -> KNOWN-ACTION
```

The 3-step plan for reformulating (UNDO (VOID PO SO)) in terms of a known action has now been successfully executed, enabling the application of RULE57:

```
                            (DURING (ROUND-IN-PROGRESS)
                                    (EXISTS C1 (CARDS)
                                            (AND [GET-CARD PO C1 LOC1]
                                                 [IN-SUIT C1 SO])))
12:11                       --- [COMPUTE by RULE57] --->
                            NIL
```

This transforms NOT's argument into the constant NIL, allowing RULE236 to be applied:

```
                      (NOT NIL)
12:12                 --- [COMPUTE by RULE236] --->
                      T
```

The initial expression has now been transformed to T, *i.e.*, proved.

7.4. Means-End Analysis for Operationalization: Summary

Means-end analysis appears to be a promising technique for automating at least some of the kind of problem-solving exhibited in the derivations. It requires a suitable representation for problem states, goal states, and differences between them, and mechanisms for finding the right operator (transformation rule) to reduce a specified difference.

Problem states are simply expressions; goal states can be abstract descriptions of expressions, *e.g.*, the variabilized patterns on the left-hand sides of rules. The search should be more focussed if the initial goal is relatively concrete rather than simply "operationalize e." Concrete goals include:

"Prove P" -- transform P into T

"Apply method M to expression e" -- transform e to match the problem statement for M

There are two basic kinds of expression differences, *semantic* and *structural*. Semantic differences involve mismatches between components of expressions. Structural differences involve components being in the wrong positions, *e.g.*, transposed. Relations useful in describing differences include:

1. f is-a g: either f = g or there is a path from f to g consisting of is-a links.
2. e has g as its head: e = (g ...)
3. e contains g: e = (... g ...), *i.e.*, g occurs as a (possibly nested) sub-expression of e
4. f is adjacent to g in e: e contains (f e_1 ... (g ...) ... e_n), *i.e.*, f has (g ...) as an argument
5. *f is directly defined in terms of g: f* is defined as (lambda (...) (g ...))
6. f is defined in terms of g: transitive closure of "directly defined in terms of"
7. f directly mentions g in its definition: f is defined as (lambda (...) (... g ...))
8. f mentions g in its definition: transitive closure of "directly mentions in its definition"

These properties are partial-ordered insofar as some imply others:

f = g
=> f is-a g

e has g as its head
=> e contains g

f is adjacent to g in e
=> e contains f, g

f = g or f is directly defined in terms of g
=> f is defined in terms of g
=> f directly mentions g in its definition
=> f mentions g in its definition

Differences can be understood in terms of expressions that fail to satisfy desired relations. For example, the pattern (cause (at ?obj ?loc)) on the left side of RULE358 can be described as a conjunction of relations between an expression e and two symbols f and g:

1. e has f as its head
2. f = `CAUSE`
3. f is adjacent to g in e
4. g = `AT`

The expression e = `(CAUSE (EXISTS C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0)))` is characterized by a conjunction of weaker relations:

1'. e has f as its head

2'. f = `CAUSE`

3'. e contains g = CARDS-IN-HAND

4'. g mentions AT in its definition

The pattern and expression both satisfy properties 1 and 2. The expression violates 3 but satisfies 3'. This indicates a *structural* difference, since its elimination involves moving g next to f. Similarly, the expression violates 4 but satisfies 4'. This indicates a *semantic* difference, since eliminating it involves expanding the definition of g. In other words, the classification of a difference as "structural" or "semantic" depends not only on what relation is *violated* but on what weaker relation may be *satisfied* in its place. Perhaps a better way to put this is to say that the choice of an operator to move an expression towards a desired pattern should depend not only on the *differences* between expression and pattern but on their *similarities, i.e.*, the relations they satisfy.

The point of the structural/semantic distinction is to classify differences according to the kinds of operators required to reduce them. Typically, structural differences are reduced by *restructuring* (§5.10) and semantic differences by *translation* (§5.8).

The choice of a rule to reduce a difference between an expression and a pattern should be constrained by exploiting information about the expression, the pattern, and the nature of the difference. *Intersection search* (§5.6) provides a way to integrate these three sources of constraint. Intersection search looks for a path between two nodes in a graph. For a semantic difference, a symbol in the expression is one starting node, and the corresponding symbol in the pattern is the other. For a structural difference, the two starting nodes are, for example, two functions to be transposed in the expression. The kinds of arcs allowed in the path are determined by the nature of the difference:

1. *Both*: Is-a links. This makes it possible to find paths from specific functions in the current expression to general concepts used in rule patterns and goals.
2. *Semantic*: An arc between the head symbols of each translation rule's left and right sides. If these are the same, multiple arcs are used between corresponding lower-level symbols of each side. RULE124 (elaboration) and RULE123 (recognition) are specially represented by an arc between each function and the head function in its definition.
3. *Structural*: Arcs for each restructuring rule. For example, a transposition rule is represented by an arc between the transposed symbols. It is unclear how to encode certain restructuring rules as arcs (associations between two symbols) in a way that preserves enough information to effectively constrain the search. For example, an arc between QUANTIFIER-SCOPE and QUANTIFIED-CONDITION would encode only partial information about the effect of the transfer rule

RULE135: (Q x (set-of y S P_y) R_x) -> (Q x S (and P_x R_x)) -- *transfer P to quantified condition*

The purpose of such intersection search is to generate a plausible *plan* (operator sequence) to solve a given operationalization problem or sub-problem. Each step in such a plan consists of applying a transformation rule. If the current expression matches the left side of the next rule in the plan, the rule is applied immediately; otherwise a subproblem is generated to reduce the differences between them. Thus the means-end analysis operates in a problem-reduction mode, executing plan steps when possible and expanding the plan when necessary to enable execution of the next step. This sort of problem-solving can be viewed as *planning in a hierarchy of abstraction spaces* [Sacerdoti 77]; the n^{th}-level abstraction of an expression is constructed by ignoring sub-expressions nested below depth n.

Chapter 8
Further Research

The research described in this dissertation can be extended in several directions, some specifically related to operationalization, others broader in scope. (§8.1) discusses aspects of operationalization that need further attention. (§8.2) discusses related aspects of *machine-aided heuristic programming* [Hayes-Roth 78a], a knowledge engineering approach based on mechanical operationalization of advice supplied by an expert or inferred from experience. (§8.3) discusses some general AI issues connected with this work. Areas for further research are summarized in (§9.2).

8.1. Further Research in Operationalization

As an operationalization engine, FOO is very incomplete; it was designed as an exploratory research vehicle rather than for production use. The hand-simulated control made it possible to do without several features that would be required otherwise, such as mechanisms for choosing which rule to apply next, keeping track of alternative paths, and deciding when to stop (§8.1.1). A practical operationalizer would need to be more time- and space-efficient than FOO (§8.1.2). Extending FOO to handle a wider range of advice would require expanding its repertoire of operationalization strategies (§8.1.3) and analysis methods (§8.1.4). A broader approach to operationalization would exploit information not used in FOO, such as knowledge about data structures (§8.1.5), details of a specific task situation (§8.1.6), and experience both in performing the task and in operationalizing previous advice (§8.1.7).

8.1.1. Control

To simplify the operationalization problem I hand-simulated control decisions about which rule to apply at each point in the derivations. A complete operationalizer would have to make such decisions automatically. Pure trial-and-error is inadequate here because of the combinatorics of the search space. The depth of this space, *i.e.*, the length of the chains of reasoning required to operationalize advice, is indicated by the length of the derivations, some of them over 100 steps long. The branching factor of the search space is also large, both because many rules are syntactically applicable at each point in a derivation, and because a transformation can be applied to any sub-expression of the current expression.

Static rule ordering proved inadequate as a means of rule selection. For example, in situations where an expression can be either elaborated in terms of a lower-level concept (§5.8.2) or recognized in terms of a higher-level concept (§5.8.3), sometimes one transformation is appropriate and sometimes the other is. In short, avoiding a combinatorially explosive search requires rule selection to be made very carefully, *i.e.*, based on as much knowledge as possible.

Means-end analysis appears to offer a promising approach for guiding the operationalization process (§7.3). The requirements of this approach are described below.

First, it must be possible to *represent operationalization goals* such as:

> Find a way to evaluate expression e.
>
> Construct a plan to achieve task goal G.
>
> Figure out how to apply method M to problem P.
>
> Reformulate expression e so as to match pattern p.

A mechanism is needed for *finding a suitable operationalization strategy* to achieve a given operationalization goal. (§2.13) characterizes FOO's operationalization strategies in terms of their usefulness in eliminating various obstacles to runtime operationality. For example, the pigeon-hole principle (§2.2) is useful in evaluating certain expressions formulated in terms of unobservable data, such as the condition "the Queen of spades is out." Finding an operationalization strategy for an expression based on analysis of why it is non-operational is a key research problem to be solved in designing an automatic operationalizer.

It must be possible to *identify differences* between the current state and an operationalization goal, such as:

> Expression e contains a sub-expression that will not be evaluable at runtime.
>
> Action A will not be directly executable at runtime.
>
> Problem P is not expressed in the same language as method M.
>
> Two occurrences of variable x in rule R correspond to unequal components of expression e.

Applying an operationalization strategy to an expression requires the ability to *find a suitable operator to reduce a difference.* This requires knowledge about the kinds of differences each method reduces. A *semantic* difference, such as a mismatch between a component of the expression and the corresponding element of the left-hand side of the rule, may be reduced by reformulating the component in terms of the concept used in the rule (§7.1.1). A *structural* difference, for example between the order of two pattern elements in a rule and the order of the corresponding components in the expression, may be reduced by restructuring the expression in a suitable manner (§7.1.3).

(§7) illustrates how intersection search could be used to find a suitable rule for reducing a given semantic or structural difference between a current expression and a goal pattern. This approach exploits information about the *similarity* or semantic relationship between the expression and the goal as well as their differences; for example, finding that the expression is indirectly defined in terms of the goal concept would suggest elaborating the expression by plugging in its definition.

If an operator with parameters is selected, it may be necessary to *decide how to instantiate the operator.* Some of FOO's rules require selecting from a set of clauses in a problem or a set of concepts retrieved from the knowledge base (§6.5). Certain types of domain knowledge could help constrain such choices. For example, a rule for undoing a conjunction selects one of the conjuncts to undo (§2.5.1). This rule could be constrained not to select a conjunct known to be immutable (§6.6.1.1).

The search control mechanism must be able to decide if it is *semantically appropriate* (*e.g.*, logically valid or sound) to apply a given operator to a given expression for a given purpose. I formulated mechanical tests of soundness or validity for many of FOO's rules, but was unable to do so for several others (§6.6). In FOO, the logical status of the current expression is often implicit. An automatic operationalizer would need to know the logical relationship between the current expression and the operationalization goals that produced it.

For example, if the goal is to *achieve* some condition C, it is legitimate to transform C into a stronger condition C', since C will be achieved if C' is. However, if the goal is to *evaluate* C, transforming it into C' leads at best to a partial solution in the sense that when C' is known to be true, so will C, but the falsehood of C' implies nothing about the truth value of C. Making logical status explicit would also require replacing the implicitly quantified variables used for brevity in some rules and concept definitions with explicitly quantified variables, or adding a scheme to mark variables as existential, universal, or global.

The ability to *pursue alternative operationalization paths* requires appropriate bookkeeping. FOO didn't need a backtracking mechanism, since I only led it down correct paths. The ability to search for an operational solution by trying multiple paths would require the ability either to undo the effects of global assumptions and variable assignments, or to confine these effects to separate contexts by means of mechanisms similar to those used in truth maintenance systems [Doyle 81] and partitioned semantic networks [Hendrix 75].

The operationalizer must *know when to stop*. Deciding whether an expression is operational requires knowledge about runtime capabilities. Encoding such knowledge requires the ability to refer to different times. For example, the trick-winner is known at the end of the trick but not in the middle. Inferring such knowledge from the task description, *i.e.*, predicting whether an expression will prove operational in the situations in which it will be used, requires a model of time, choice, observation, and memory. For example, an opponent's choice of what card to play becomes known when the card is played. Thus once a trick begins, the suit led will be known, but not (in general) whether the trick will have points, since this depends on cards not yet played.

FOO's evaluation of operationality is limited to a procedure that takes an expression and returns a list of functions which, if executable, guarantee that the expression is operational, as shown in Appendix E. The problem of testing operationality is further complicated by the fact that operationality is relative: the more often a procedure can be executed, and the more accurate its results, the more operational it is. In practice operationality might have to be evaluated empirically, by applying the procedure to actual task situations and seeing how often it produces the desired results.

8.1.2. Efficiency

FOO's design emphasizes generality at the cost of efficiency. A useful system would need to be more efficient in time and space.

FOO takes over 300K on a PDP-10, partly because it holds the entire derivation in core. An obvious remedy is to store only the information required to continue the search or backtrack to an earlier point: the current stack of operationalization goals, plus information about the approaches that have been tried on each one. However, a practical operationalizer might well need large amounts of memory, for example more than the address space of a PDP-10. This suggests using a machine with virtual memory and a large address space -- at least until multi-million-word primary memories become sufficiently cheap to support the escalating demands of AI research.

It proved necessary to disable a feature of FOO that listed all the rules applicable to the current expression, because it took too long to test the rule conditions (*i.e.*, longer than I was willing to sit and wait). Automatic rule selection, or even interactive rule selection (where the system recommends a rule and the user accepts it or asks for another), would require a more patient user or a more efficient implementation. Means-end analysis (§7) would help here by considering only rules relevant to current operationalization goals, rather than identifying all syntactically applicable rules.

Compiling frequently-used rule sequences into procedures would extend the range of possible solutions that could be explored given specified time and space resources. In particular, simplification rules could be incorporated into a quasi-canonicalization procedure applied after each transformation (§5.2.5). Special-purpose inheritance mechanisms could be added to make certain kinds of deductions requiring sequences of rule applications in FOO, such as verifying that an intensionally described object (or set) is a member (or subset) of a particular class. Such mechanisms would require explicit representation of type information encoded in FOO as domain-specific "fact rules" (§1.3.3).

Specialized methods could exploit a frozen (as opposed to open-ended) set of concepts or canonical representation. For example, FOO's intersection search procedures (§5.6) circumvent the detailed case analysis that might otherwise be required to reject certain impossible propositions. Their validity is based on some completeness assumptions about the knowledge base (§6.5.1).

8.1.3. Representation of Operationalization Methods

FOO has only a few operationalization methods. A useful operationalizer would require many more and might profitably exploit existing general techniques for planning [Sacerdoti 77] [Carbonell 78] and automatic programming [Manna 79] [Barstow 77].

8.1.3.1 Acquisition of Operationalization Methods

Developing a large repertoire of general operationalization methods would be easier if they could be acquired in the same manner as task-specific concepts and heuristics. Ideally, it should be possible to improve an operationalizer simply by telling it about a new concept, such as the distribution of cards in a suit (§6.5.5); the concept could then be used to operationalize advice. Assimilating new operationalization strategies expressed in the same representation as domain knowledge is an interesting and difficult problem.

A useful step towards this ideal is the development of a declarative conceptual representation for the kinds of knowledge involved in understanding a method: its scope of application, its informational requirements, and the factors determining the accuracy of its result. Such a representation would make an easier target for mechanical interpretation of informal input than the transformation rule format in which FOO's methods are hand-coded. Since an operationalization strategy is a way to implement a desired capability in terms of the operations available to a given task agent, a representation of such a strategy may need to refer to a model of the task agent. The simple task agent model presented in (§1.1.1) is useful in explaining some operationalization strategies, but would have to be extended to deal with strategies for reducing computational costs, such as methods for speeding up a heuristic search (§3.4).

8.1.3.2 Representation of AI Methods

FOO's representation of the heuristic search method was inspired by Newell's flow graph schemata for the weak methods [Newell 69]. Moore encoded the flow graph for heuristic search as a semantic network in MERLIN, and instantiated it by hand to form an operational version of the Logic Theorist [Moore 74]. This dissertation extends Moore's work by showing how such an instantiation process might be mechanized (§3). Encoding other AI methods in a form conducive to mechanical application is a challenging and worthwhile research problem.

8.1.4. Analysis Methods

Analysis makes contact between general operationalization ideas and specific problems. Analysis subsumes theorem-proving in the sense that the latter can be defined as symbolically transforming a proposition into a truth constant, while the former reduces an expression (not necessarily boolean) into one that's simpler or easier to operationalize (not necessarily constant). Moreover, theorem-provers are restricted to logically correct transformations, while analysis may use heuristically useful transformations whose results are occasionally incorrect but usually lead to good performance. Researchers in program verification have observed that verification conditions rarely reduce to true or false, and that mechanical reduction of a program's correctness precondition can help pinpoint errors in the program or its specifications [Nelson 79].

FOO's analysis rules were formulated to handle the particular operationalization problems undertaken, sometimes in a somewhat *ad hoc* fashion. Their purpose is to *indicate* (not necessarily *provide*) analysis capabilities useful in operationalization. A practical general-purpose operationalizer would need a more powerful analysis engine. Several steps would help toward this goal:

1. Formulate soundness and validity tests missing in some of FOO's rules (§6.6).
2. Generalize FOO's rules to apply the same transformations to a wider range of expressions.
3. Develop new rules for additional analytic transformations.
4. Develop analysis methods that exploit semantic relations among task concepts (§6.1).
5. Incorporate systematic proof procedures based on special-purpose representations (§2.10.3) (§5.6) (§6.1).

8.1.5. Knowledge About Memory

Several researchers have worked on the problem of mechanically designing or selecting an appropriate data structure for a given problem [Rovner 76] [Low 78] [Barstow 77]. FOO addresses only the first step of this process, namely identifying what *information* is necessary or sufficient to calculate a given quantity (§2.4). Some of the "operational" procedures derived using FOO contain *historical references* to such quantities as the number of times a particular kind of event has taken place during a specified time period (§2.4.3). Implementing such a solution requires a mechanism for monitoring and counting such events. FOO has a search refinement rule for recognizing that a particular function can be precomputed and stored in a table before the search (§3.5.3.2), but FOO's applicative representation provides no way to refer to such tables. A more complete treatment of time-based concepts would require an *explicit representation of memory processes and data structures.*

8.1.6. Dynamic Operationalization

Dynamic operationalization reduces a general operationalization problem to an easier one by exploiting an additional source of knowledge -- the features of a specific task situation. For example, if the King of diamonds has been led, the advice "avoid taking points" can be operationalized as "underplay the King of diamonds" (§5.4.4). Dynamic operationalization is to situation-independent operationalization as planning is to automatic programming (§9.1.1): it solves an easier problem but produces a less general result.

A practical operationalizer might try to do as much reasoning as possible in a situation-independent manner, and then use dynamic operationalization to find special-case solutions for remaining problems. For example, the advice "avoid taking points" can be partly operationalized, without reference to a specific trick, as "underplay one of the other cards in the trick." It is not possible in general to predict "the other cards in the trick." However, it is often possible to dynamically operationalize "underplay another card in the trick" in the context of a particular trick, for example by underplaying a card that has already been played or that one knows will have to be played. In situations lacking such a special-case solution, it may be possible to revert to a general but less effective solution, *e.g.*, "play a low card." Automatically discovering such *combinations of general and dynamic operationalization* is a problem for future research.

8.1.7. Applications of Learning

Sources of data from which an operationalizer could learn include:

D1. Description of the task domain.

D2. Advice for performing the task.

D3. Observations made in actual or hypothetical task situations.

D4. Operationalized advice.

D5. Derivations of operational solutions.

D6. Attempts to operationalize advice, both successful and unsuccessful.

Methods that could be used to learn from this data include:

L1. *Analysis*: logical or heuristic deduction of new knowledge from old.

L2. *Learning from examples*: generalizing from individual events.

L3. *Learning from experience*: gradual accrual of knowledge from a series of events.

Applications of learning in operationalization can be characterized according to which *methods* are applied to what *data*:

[D1, D2, L1] FOO lies squarely at the *analytic* end of the *analytic*-to-*empirical* spectrum of inference methods. This rather narrow approach to operationalization could be extended by exploiting *empirical* paradigms for knowledge acquisition, as illustrated by the succeeding proposals for further research.

[D1, L1] plus [D3, L3] The marriage of *analytic* and *empirical* techniques appears promising. For example, if analytic techniques predict that one quantity is an increasing function of another, empirical data gathered from task experience can be used to estimate the actual function (§2.8.4). In general, learning important relationships among task components would improve the ability to operationalize. For example, it is useful to know that PLAYER-OF is the inverse of CARD-OF and that a player who is void in a suit stays void for the rest of the round (§6.6.1.1). Such facts could be induced by analysis of the task description, by induction from task experience, or by some combination of the two. A smart learner would look for semantic relationships exploited by analysis methods, such as immutability, inverses, and homomorphisms. Techniques for automated discovery of such relationships have been used to find connections between mathematical concepts [Lenat 78].

[D3, L2] Teaching by example is a valuable mode of tutorial instruction not exploited in FOO. It is often easier to communicate a concept by an illustrative example than by trying to state it abstractly. Accepting advice in this form would require the ability to identify the criterial features of the elements in the example and infer the relevant abstraction. The idea of *exemplary programming* is discussed in [Waterman 78].

[D3, D4, L3] Analytic methods are often inadequate to predict the operationality of a heuristically derived implementation of a piece of advice (§8.1.1). For example, how often will the plan "play a low card" satisfy the goal "avoid taking points?" A much easier way to answer such questions is to use standard reinforcement techniques: increase the estimated worth of a plan each time it works, and decrease it when it fails. This is similar to keeping track of the plan's average effectiveness over time, except that decreasing a plan's worth will cause it to be used less frequently thereafter when an alternative plan is available. (If there isn't, such a decrease will enhance the priority of developing one.)

[D2, L3] The same approach could be used to evaluate the effectiveness of a piece of advice in achieving higher-level task goals, *e.g.*, how often does following the advice "avoid taking points" actually lead to winning the game?

[D4, D5, L2] A solution generated for a given problem can sometimes be generalized to cover other problems as well. For example, the plan derived for deciding whether the Queen of spades is out (§2.2.1) can be generalized to work for any card. This can be determined by verifying that none of the steps in the derivation depends on any properties specific to the Queen of spades other than the fact that it's a card. Generalizing the original plan would consist of substituting a variable `C1` for the constant `QS` and making `(IN C1 (CARDS))` a precondition of the plan. In general,

```
To generalize a plan P containing a constant c,
Replace c with a variable x and
Add as a precondition that x must satisfy all tests applied to c in the course of deriving P.
```

Note that this technique relies on *inspection* of the derivation without additional analysis. The idea of generalizing a plan by suppressing irrelevant details was used in STRIPS [Sacerdoti 74]. The computational resources required to generalize a derived solution are well-spent if doing so eliminates the need to solve similar problems in the future.

[D4, L2] A challenging problem is to predict the relative usefulness of different operationalization methods for a particular piece of advice. A simple approach is to compile a record of each method's average performance along an appropriate set of dimensions: how expensive is it to apply? how often does the result work? how expensive is the result to use? For example, a key measure of a heuristic search refinement rule (§3.4) is how much it speeds up the search -- or slows it down.

[D4, L2] A more sophisticated approach to evaluating operationalization methods would compare solutions derived using alternative methods and try to discover individual factors predicting the differences between their performance. For example, the cost-effectiveness of a rule for compiling a constraint out of a search by precomputing a function and storing it in a table (§3.5.3.2) might be found to depend on such factors as the depth and branching factor of the search space, the frequency and cost of computing the function, and the storage cost for the table. A formula based on these factors could be used in deciding whether to apply the rule to subsequent problems. Synthesis of the formula and evaluation of the factors could draw on both empirical and analytic methods. Related research on machine comparison of alternative algorithm refinements includes work on concrete computational complexity [Kant 79] [Wegbreit 75].

[D6, L3] An interesting possibility for learning about operationalization itself is to treat the operationalizer's own deliberations as data. One scheme along these lines would learn to apply a method only to problems where it is likely to work. Suppose successful attempts to apply a particular method are empirically observed as tending to follow a common scenario, whereas unsuccessful attempts fail to get past some point in this scenario. Analysis of both the successful [Mitchell 81] and unsuccessful [Hayes-Roth 81b] attempts could identify tests that predict when the method will work (§2.6.4.1). These tests could then be incorporated as conditions for applying the method.

8.2. Further Research in Machine-Aided Heuristic Programming

Mechanizing the operationalization process is a subproblem of building a *machine-aided heuristic programming system* capable of translating informal task descriptions and advice into effective behavior. This section briefly discusses other problems involved in implementing the main phases of machine-aided heuristic programming:

1. *Interpret* informally expressed advice and domain knowledge into an unambiguous representation (§8.2.1).

2. *Operationalize* advice as effective plans and procedures.
3. *Integrate* multiple, possibly inconsistent pieces of advice (§8.2.2).
4. *Apply* advice to runtime task situations (§8.2.4).

[Hayes-Roth 78a] describes several of these problems in greater detail, summarizes some approaches explored by other researchers, and proposes some additional approaches.

It is important to note that the interpretation, operationalization, integration, and application phases in a machine-aided heuristic programming system would overlap in time, rather than follow a strict pipeline flow of control. For example, ambiguities in advice might be resolved in the context of applying it to a particular task situation; a similar approach proved successful in SAFE, where vague references were resolved in the context of the variables active at runtime, and incorrect interpretations were discarded when they led to runtime errors during symbolic execution [Balzer 77]. These phases have been separated for purposes of discussion, and operationalization has been considered in isolation from the others in order to simplify the problems addressed in the dissertation.

8.2.1. Interpretation of Informal Advice

Conversion of expertise into behavior involves translating informally expressed advice into a precise representation. Such advice may contain constructions like "taking points," "lead a losing heart," and "safe suit." These constructions violate the well-formedness property assumed in FOO.

To illustrate this point, consider the concept `TAKE`, defined in FOO as

```
(LAMBDA (P C) (MOVE C POT (HAND P)))
```

One instance of this concept is `(TAKE ME (WINNING-CARD))`. The mapping from the actual arguments `ME` and `(WINNING-CARD)` to the lambda variables `P` and `C` is *positionally determined* since they are specified in the same order, so there is no problem in identifying the arguments with the lambda variables.

Slot-related information might be useful in interpreting expressions with missing or mistyped arguments. For example, consider the informal construction `(TAKE POINTS)`. Knowing that the lambda variable `C` of `TAKE` is card-valued and that `POINTS` is a function on cards might suggest that `POINTS` instantiates `C` rather than `P`. `P` might be filled in by storing `ME` as its default value. However, further inference would still be required to interpret the informal input (TAKE POINTS) [Mostow 79b] as

```
(TAKE-POINTS ME) = (FOR-SOME C1 (POINT-CARDS) (TAKE ME C1))
```

The most closely related research on this problem is the SAFE project [Balzer 77], involving automatic translation of informal specifications into working code. The information provided by an actual task situation can provide useful context for interpreting informally expressed advice; for instance, alternative interpretations of ambiguous advice can be tried out in a specific situation to see which one makes sense or leads to the best outcome [Balzer 77].

8.2.2. Integration

Integrating different pieces of advice is a crucial and difficult problem. I simplified the operationalization problem addressed in FOO by assuming that advice could be operationalized before integrating it with other advice. An evaluation function with terms for each piece of advice can sometimes be used to integrate independently operationalized advice [Berliner 79]. Fuzzy applicability conditions for each piece of advice can be operationalized as coefficients on the various terms [Berliner 79]. A challenging problem that does not appear to fit this simple scheme is to integrate advice quantified over goals (§8.2.3) or other pieces of advice, *e.g.*:

"Infer that a player who did something did it for the same reason you would."

8.2.3. Using Knowledge About Goals

FOO lacks an explicit model of actors' goals. Such a model would be very useful. For example, it would make it possible to operationalize advice of the form "To achieve X, do Y." Such advice could be used to interpret other agents' behavior based on knowledge about goals and plans:

G1. Assume another's action is motivated by the same reason you would do it.

G2. Assume the preconditions of a plan are satisfied if someone seems to be following it.

G3. Assume others will pursue a goal using the same plan you would.

Such knowledge can assist both evaluation and planning. For example, G1 suggests that a player who leads spades is trying to flush the Queen. G2 implies that this player does not have the Queen. G3 predicts that the player will cooperate in an effort to keep leading spades until the Queen is played. A program that uses an explicit representation of actors' goal structures both to *plan* and to *interpret* behavior is described in [Carbonell 78]. The idea of an *invertible representation* supporting both these uses is developed in [Hayes-Roth 78a] and [Mostow 79b].

A goal model would also be required to determine the *implicit purpose* of a piece of advice. For example, a human player who receives the advice "flush the Queen" will realize that the purpose of doing so is to eliminate the danger of taking it. This ability could be invaluable both in interpreting and integrating informally expressed advice (§8.2.1) (§8.2.2). For example, the advice "flush the Queen" could be interpreted as "flush the Queen *without taking it*" based on an analysis of its implicit purpose; this would prevent the mistake of leading the Ace to flush the Queen. Similarly, goal-based analysis of "avoid taking points" indicates that this advice should be disregarded when shooting the moon. Building a system to understand loosely phrased advice by relating it to known goals would make a challenging and interesting dissertation project.

8.2.4. Runtime Mechanism

The runtime mechanism assumed in the dissertation is an unimplemented but straightforward extension of LISP EVAL. The basic extensions include the ability to cache computed values [Lenat 79] [Marsh 70], to try alternative partial definitions, and to exploit annotations about necessary or sufficient conditions in specified situations (§2.6.3).

8.3. Further Research in AI

In developing FOO, I encountered some general representation problems of AI. These include representation of imprecise concepts (§8.3.1) and of the idea of knowing something (§8.3.2).

8.3.1. Representation of Conceptual Knowledge

The power of lambda expressions for representing the meaning of natural language is examined in [Hobbs 78]. The dissertation shows how to encode certain familiar classes of concepts (*e.g.*, different kinds of adjectives) as lambda expressions of appropriate forms (§1.3.1.3). However, abstract concepts like "good" and "danger" remain difficult to encode in this representation. Doing so appears to require an explicit model of agents' goal structures (§8.2.3). For example, a thing is "good" for an agent whose goals it helps, and "dangerous" for one whose goals it threatens. One way to avoid formulating precise definitions for such concepts is to use assertions and partial definitions: instead of trying to find a complete definition of "danger", simply assert that certain things are dangerous, and relate dangerous things to other concepts, *e.g.*, "avoid dangerous situations."

8.3.2. Knowledge about Knowing

FOO has no explicit model of the idea of an agent knowing a piece of information. This is a hard epistemological problem on which some theoretical work has been done [McCarthy 77] but much remains. Some model of the "knowing" relation is required to encode and understand advice like

"Lead a suit to see who's void in it."

"Conceal your intentions."

Chapter 9
Conclusion

This chapter places the dissertation in the context of past, present, and future research. (§9.1) reviews prior research related to operationalization and explains how it differs from the work reported here. (§9.2) lists some important problems requiring further research. Finally, (§9.3) evaluates the dissertation's own research contributions relative to the goals set forth in (§1).

9.1. Related Research

This section discusses relevant prior research. (§9.1.1) defines some terms used in this discussion: *problem-solving, constraint satisfaction, planning,* and *automatic programming.* (§9.1.2) considers operationalization in terms of each of these concepts. (§9.1.3) discusses previous work on the broad problem of *machine-aided heuristic programming,* which includes operationalization as a sub-problem.

9.1.1. Problems, Constraints, Plans, and Programs

Operationalization is closely related to the often -- and loosely -- used AI terms *problem-solving, constraint satisfaction, planning,* and *automatic programming.* Intelligent discussion of the relationship among these concepts dictates an attempt to define them first.

Problem-solving [Newell 60] is a very general term denoting the activity of finding or constructing some entity (a *solution*) satisfying a given set of requirements (a *problem*). This solution is the intended result of the problem-solving process, *i.e.*, the purpose for undertaking the problem-solving process; the particular steps taken to reach it may be of interest but are not part of the solution. This distinction is complicated by the fact that the same problem-solving activity can be viewed at more than one level. For example, consider the problem of evaluating an expression, *i.e.*, tranforming it into a constant that denotes the same value. At this level, the solution is the constant value itself.

Now consider the problem of finding an equivalence-preserving transformation sequence from the expression into a constant. At *this* level, the solution is the transformation sequence.

Constraint satisfaction [Fikes 68] is a particular kind of problem-solving where the problem consists of a set of assertions about (*constraints* on) a collection of variables, and a solution is a set of variable assignments satisfying all the constraints.

Planning [Sacerdoti 77] finds a sequence of actions (a *plan*) which when applied to a specified initial state will produce a desired state (*goal*), represented as a description of a particular state or as a predicate on states. The initial state is an argument of the planning process, so the plan need only work for that state. Typically the range of possible actions is defined by a given set of state-to-state mappings (*operators*). The intended result of the planning process is the plan, not the final state. This distinction can be subtle in programs that solve problems by constructing and executing plans, especially when plan construction and execution are interleaved. The overall activity constitutes problem-solving, but only the plan construction phase, *i.e.*, selection of an action sequence, constitutes planning.

Automatic programming generates a program whose execution will achieve a desired effect, *i.e.*, produce a state satisfying a desired relationship (*specification*) with the initial state, assuming the initial state satisfies a given set of assumptions (*precondition*). The initial state is unknown at program generation time; it is an argument of the generated program, where it is typically represented by the program's input variables. The program must work for any initial state satisfying the precondition. In this sense, automatic programming is more general than planning, although the distinction is blurred in systems that construct plans and then generalize them [Sacerdoti 77] (§8.1.7).

9.1.2. Prior Research Related to Operationalization

Operationalization is a form of *problem-solving*. In FOO, an operationalization problem is an expression describing a goal to be achieved or a quantity to be evaluated in the course of performing some task. A solution is an expression whose execution achieves the goal or computes the value of the quantity.

Operationalization is also related to *constraint satisfaction*: the thing to be constrained is the behavior of a runtime mechanism performing a task, and a constraint is a piece of advice restricting this behavior. The task imposes further structure on this behavior by making it fit a specified computational process, such as the sequence of legal play in Hearts or the left-to-right generation of tones in composing a *cantus firmus*. However, an operational solution is more complex than a collection of variable assignments.

Operationalization in FOO can be viewed in two very different ways as *planning*. At the level of the task domain, the operators are the actions the task agent can perform and the observations it can make, the states are task situations, and the problem is to construct an operational plan composed of such actions to achieve a specified condition or evaluate a specified quantity. This differs from the definition of planning given above in that the initial state is not an input to the planning process; the plan is supposed to work in a variety of task situations. The inherent capabilities of the task agent may be quite limited: for example, a Hearts player cannot directly observe opponents' cards or control their play. Work in problem-solving, planning, and constraint satisfaction typically assumes a single agent in an environment where everything can be observed. Exceptions include programs for understanding goal-based stories [Schank 77] and international relations [Carbonell 78].

At the level of the operationalization process itself, the states are not task situations but expressions. The initial state is a non-operational expression, and the goal is to make it operational. The operators are FOO's transformation rules. The problem is to find a rule sequence that transforms the initial expression into an operational form. Such a sequence is a plan for operationalizing the given expression, and need not work for any other. Operationalization consists of finding *and executing* such a plan.

Finally, operationalization satisfies the above definition of *automatic programming*: the specification describes a goal to be achieved or a quantity to be evaluated, and the problem is to generate a program whose execution achieves the goal or computes the value of the quantity in a wide class of task situations. Both the specification and the program are represented as expressions.

Several researchers in automatic programming have worked on the problem of transforming an abstract program specification into working code [Low 78] [Barstow 77]. The specification is actually a program in a high-level language that allows abstract types, *e.g.*, "set," and corresponding abstract operators, *e.g.*, "union"; the language used for this purpose in PSI actually had a working (although of course rather inefficient) interpreter [Green 76]. Implementing such a specification can be viewed as operationalizing it for a task agent capable of interpreting the target language. It involves translating the abstract objects in terms of code-level primitives, *e.g.*, "linked list" and "append," sometimes via a process of *stepwise refinement* through intermediate levels of abstraction, *e.g.*, [illegible] set."

This process can be viewed as designing or selecting a *representation* (data structure and associated operators) for each abstract object in the program. The efficiency of the implementation depends on the cost of the operators in the chosen representation, and on how often they are applied to each object [Low 78] [Kant 79]. Korf's work [Korf 80] is similar but translates a representation into another at the same level; for example, "list" and "append" might be translated into "bit vector" and "bitwise OR." FOO differs from the work described above in that it lacks knowledge about data structures *per se*; some of FOO's rules decide *what* information to remember (§2.4) (§3.5.3.2), but not *how* to store it (§8.1.5).

A key difference between operationalization and what is usually called automatic programming has to do with the relationship between the specification and the program generated to satisfy it. In traditional automatic programming, the program must be a *correct* implementation of the specifications, *i.e.*, it must produce exactly the specified result whenever the program's precondition is satisfied. In contrast, operationalization produces an *operational* implementation that may not always produce the specified result, or may produce an approximation to it (§1.1). The *fallible* and *approximate* methods permitted in operationalization can be used in situations where the *exact, formally correct* methods of automatic programming fail. These situations are typical of tasks requiring heuristic knowledge. For example, no Hearts program can infallibly "avoid taking points," but this advice can readily be *operationalized* as "play a low card."

9.1.3. Prior Research Related to Machine-Aided Heuristic Programming

The operationalization problem formulated in this dissertation arose from the desire to build a general *machine-aided heuristic programming system*. The various problems involved in building such a system are described in [Hayes-Roth 78a], and the paradigm is developed further in [Hayes-Roth 80]. The work reported in this dissertation is related to many other research efforts but differs from them in its direction.

Most automatic programming research has sought to produce a program given a *formal specification* of its input-output relations and nothing else. The contrasting approach taken here views operationalization as a part of *machine-aided heuristic programming* (§1), which seeks to produce an effective implementation for an *informally* specified task, given *knowledge about the domain* of the task and *heuristic advice* on how to achieve the desired behavior. This additional knowledge provides a potential source of power typically unexploited in automatic programming.

The SAFE [Balzer 77] and PSI [Green 76] systems are partial exceptions to these general statements about formal input, domain knowledge, and advice. PSI accepted English input, but required problems to be expressed in terms of rather formal built-in concepts, such as the notion of a correspondence. SAFE accepted parenthesized English specifications containing informality in the form of ambiguity, implicit operations and operands, and imprecise references. Both programs used domain knowledge. However, neither of them were able to accept heuristic advice about performing the task.

Various other systems have been built to accept *informal, natural language task specifications* [Davis 77b] [Heidorn 74] [Novak 76] [Simon 77]. These systems are either constrained to a single domain or limited to a shallow understanding of their input. The specifications they accept are statements about the task, not general advice about how to perform it.

The idea of an *advice taker* has been around for some time. In [McCarthy 68], "advice" consisted of useful facts encoded as logical axioms to be used by a theorem-prover in the formal derivation of a plan. In contrast, machine-aided heuristic programming focuses on accepting informal heuristics and using them to guide task behavior.

Exemplary programming [Waterman 78] seeks to infer a general algorithm for solving a problem by observing *how* to solve it in one or more specific instances, without understanding *what* the problem is. Machine-aided heuristic programming uses both a description of *what* is desired and advice about *how* to achieve it [Hayes-Roth 78a] [Mostow 78].

A growing body of research is concerned with *learning general principles from examples* [Buchanan 78] [Hayes-Roth 78c] [Dietterich 81] [Vere 78], *observations* [Soloway 77], *and experience* [Lenat 78] [Mitchell 77] [Waterman 70]. In contrast, machine-aided heuristic programming is a form of *learning by being told*, or more precisely, *learning by discovering how to do what you're told*: it seeks to apply such principles once they have been described by an expert.

Heuristic programming seeks to achieve expert performance on a task by incorporating in a program the knowledge of an expert practitioner [Balzer 66] [Buchanan 78] [Buchanan 69] [Carnegie-MellonUniversity 77] [Davis 77a] [Duda 78] [Feigenbaum 77] [Greenblatt 67] [Lenat 78] [Martin 77] [Stefik 80]. Such *knowledge engineering* typically uses a human programmer as the interface between the expert and the computer. This interface has high latency time and low bandwidth: it can take a considerable amount of initialization for the expert and the programmer to develop a common vocabulary, and the reimplementation cost of modifying the program to incorporate additional knowledge is often prohibitive.

This knowledge assimilation bottleneck has motivated efforts to develop systems which can communicate with experts in terms of their own domain concepts [Nii 79] [Davis 77b] [Hewitt 75] [Martin 77] [Zobrist 73]. So far, the only such systems are either domain-specific or very limited in the form of knowledge they can accept, *e.g.*, MYCIN-style rules and MOLGEN units rather than arbitrary, informally expressed suggestions about the system's own problem-solving behavior. The machine-aided heuristic programming approach seeks to extend the form and range of knowledge such a system can accept.

Several existing programming languages are *extensible*. However, new constructs must be defined by procedural composition of old ones, and the details of the composition must be explicitly spelled out. The goal of machine-aided heuristic programming is to allow experts to communicate informally in terms of domain concepts, without having to specify the details involved in applying one concept to another, rather than having to code their knowledge in a formal language.

It should be emphasized that this dissertation is not about *game-playing* programs. In fact, FOO lacks the mechanisms required to actually play a game of Hearts or compose a *cantus firmus*, since doing so was unnecessary for the purposes of the dissertation. Hearts was simply an experimental vehicle well-suited to investigating the general phenomena of interest. However, two game-playing programs seem relevant. The General Game Playing Program [Williams 72] is sometimes described as taking advice about how to play. GGPP was a high-level, game-oriented programming language with a general matcher for finding instances of game patterns defined in terms of built-in primitives. Input to GGPP consisted of a game description and an *algorithm* (not advice) for how to play it. Waterman's poker program [Waterman 70] improved its performance based on feedback from the results of applying its productions for bidding; one might view such feedback as advice. However, the poker task was built into the program rather than input to it, and the program could only learn productions expressed in terms of range restrictions on built-in variables.

9.2. Problems for Future Research

FOO addresses a small part of a large problem whose solution calls for many additional techniques and sources of knowledge. (§8) describes some of the remaining problems and proposes detailed approaches to them; these problems are summarized briefly below.

Many kinds of operationalization knowledge are used in the construction of intelligent programs, *e.g.*:

Operationalization strategies

AI weak methods

Analysis techniques

Knowledge about data structures

FOO represents only a tiny fraction of this large body of knowledge. Formalizing such knowledge and encoding it in a form suitable for mechanical application will require considerable work. In particular, the purely analytic approach embodied in FOO should be considerably enriched by exploiting empirical methods and sources of information not used in FOO (§8.1.7).

Operationalization is closely related to other problems involved in machine-aided heuristic programming, many of which have received attention from other researchers:

Interpret informal input

Integrate multiple goals

Understand goal-oriented behavior

These connections are not addressed in FOO but should be explicitly considered in future research. Building a practical machine-aided heuristic programming system will require formalizing and encoding a large repertoire of techniques used in natural language understanding, knowledge engineering, and program design, but if successful will provide a corresponding large productivity payoff in the development of intelligent programs.

9.2.1. Some Interesting Dissertation Topics

Several of the problems described above would make interesting dissertation projects:

Interpret, operationalize, and integrate advice in terms of the goals it explicitly mentions or implicitly affects (§8.2.3). For example, infer that the purpose of the advice "flush the Queen" is to remove the danger of taking it, and fill in the missing qualifier: "flush the Queen *without taking it.*" Similarly, infer that the purpose of the advice "get void" is to enable sluffing (discarding dangerous cards) when the suit is led; use this knowledge to discount the desirability of getting void in a suit unlikely to be led. Finally, operationalize vague terms --- like "dangerous" and "desirable" in the preceding sentence -- as implicit references to whatever goals are relevant in the current task situation.

Represent AI methods in a form conducive to their mechanical application. The formulate-and-refine approach appears promising. Tappel has devised some general algorithm-improving operators represented as local transformations on data flow graphs [Tappel 80]. This work should be combined with refinement techniques based on domain knowledge and analysis like those described in the dissertation (§3.4).

Automate the control of the operationalization process (§8.1.1). Means-end analysis looks like a good approach to the problem-solving issues avoided in FOO (§7). A fruitful research plan might aim first at automatic application of high-level operationalization strategies selected by a user, perhaps from a machine-generated list of suggestions. If enough of the problem-solving can be automated, interactive operationalization could prove useful as a knowledge engineering technique. Fully automatic operationalization will require additional research, such as the development of a model of runtime task capabilities.

Learn concept features (like reflexivity (§5.3)) *and semantic relations among concepts* (like homomorphisms (§6.1.2)) *and develop analysis methods to exploit them.* Such methods can quickly make inferences about an expression that would otherwise require cumbersome analysis based on expanding its definition in terms of primitive entities. The properties they exploit could be learned by a combination of empirical observation [Lenat 78] and analysis of the task description and concept definitions (§8.1.7). Possibly the concept properties and analysis methods would feed on each other, with new properties enabling further analysis enabling the discovery of higher-level properties.

Automate the acquisition of operationalization strategies. What inferences can be made upon receiving the definition of a new concept, *e.g.*, suit distribution, that will enable it to be used in operationalizing advice (§6.5.5)? How can an operationalization strategy be refined based on the experience of trying to use it (§8.1.7)? Just as the conversion of task heuristics into task performance is a bottleneck in the construction of intelligent task agents, the acquisition of operationalization knowledge may prove a bottleneck in the construction of intelligent operationalizers, and therefore equally deserving of mechanical assistance.

9.3. Evaluation

The appropriate criterion for evaluating the dissertation is how well it supports its thesis (§1.1.3):

> *There exist general operationalization methods that can be stated independently of any specific task domain, although applying them generally requires domain-specific knowledge.*

The strategy used to gather evidence for this thesis was to

Represent a diverse set of general operationalization methods and apply them mechanically.

The dissertation research should therefore be judged according to the following criteria:

1. The *formal representation* of methods in FOO (§9.3.1)
2. The *diversity* of FOO's methods (§9.3.2)
3. The *generality* of FOO's methods (§9.3.3)
4. The *mechanical applicability* of FOO's methods (§9.3.4)

In particular, *coverage* is not an appropriate evaluation criterion. The goal of encoding a broad collection of methods is a very ambitious one, and the dissertation only scratches the surface. Clearly there exist many general operationalization methods besides those encoded in FOO; extrapolation from FOO suggests that it should be possible to encode others in a similar manner.

There are other perspectives from which the dissertation can be evaluated. From the broad perspective of *knowledge engineering*, it should be judged by its contributions toward the goal of building a machine-aided heuristic programming system. The feasibility of this goal has been argued elsewhere [Hayes-Roth 78a]. The representation and mechanical application of operationalization methods in FOO provide evidence for the feasibility of an important subgoal: automatic operationalization. Many of the other problems involved, such as interpretation of informal natural language input, have received attention from other researchers [Balzer 77]. Prototypes of the machine-aided heuristic programming paradigm, with hand simulation of some of the inference processes, have been built to acquire, apply, and refine knowledge in various task domains [Hayes-Roth 80].

Psychological validity is not a necessary condition for the success of the dissertation, but certain aspects of FOO are by design consistent with psychological intuition. In particular, some of the derivations incorporate a *solve-and-refine* strategy I observed in my own behavior when operationalizing advice by hand: a crude initial solution is contructed and then *refined* by taking more information into account (§2.3.1.5) (§3.4), for example by searching the knowledge base for a concept of a specified form. Given a mechanism for performing such a search, the ability to solve a particular kind of problem can be dramatically improved just by acquiring a suitable concept in terms of which to view the problem (§6.5.5). This suggests a way for systems to extend their range of competence simply by being told new ideas, an intelligent trait sometimes exhibited by humans.

9.3.1. Formal Representation of Operationalization Methods

The *formal representation* of various methods for operationalization and analysis is a central contribution of the dissertation. In particular, it formalizes certain classes of operationalization problems as equations and identifies some analysis methods useful in solving them:

1. e = $h(f(x_1, ..., x_n))$ -- reformulate e in terms of concept f (§4.2)
2. e = h(v) -- reformulate e as a function of quantity v (§4.2)
3. d = D_v(e) -- approximate e as a function of quantity v (§2.8)
4. P => Q -- find a necessary condition for P (§2.6)
5. Q => P -- find a sufficient condition for P (§2.6)

The unknowns are shown in italics. Note that the unknown h in equations 1 and 2 is a function; it represents the relationship to be found between two givens.

The formalization of reformulation problems and techniques for their solution extends previous work on *multiple views*: "reformulating e in terms of concept f" can be called "viewing e as an f." Such work has primarily addressed the problem of *representing* different perspectives on an entity [Brachman 79]. Efforts at *inferring* a perspective -- figuring out how to view an X as a Y -- have largely relied on heuristic matching and inheritance mechanisms [Moore 74] [Bobrow 77]. In contrast, FOO *formalizes the problem* of viewing one concept as another, provides analytic methods for *discovering* a relationship between them based on their definitions, and *explicitly represents* both the relationship and the assumptions made in the course of deriving it.

The dissertation identifies *general scenarios* for applying certain operationalization strategies and analysis methods, including:

Problem separation (§5.7)
Finding a necessary or sufficient condition (§2.6.4.1)
Partial matching (§5.3.1)
Heuristic search (§3.3)

These scenarios reduce a series of differences between an initial problem and the problem statement of the particular method until the method can be applied; some scenarios incorporate a subsequent phase in which a crude initial solution is *refined* (§1.6).

The dissertation contributes to the understanding of various existing analysis methods (*e.g.*, partial matching (§5.3) [Hayes-Roth 78b]) by representing them in a *common transformational paradigm.* This uniform model for the description of different problem-solving methods should help future researchers to *elucidate the logical status* of various methods (§5.11) and to *combine* them in solving problems.

In addition, the dissertation *extends* some previously described methods. For example, a previously described calculus of functional dependence [DeKleer 79] is augmented to include a multiplication operator based on the chain rule for differentiation (§2.8).

9.3.2. Diversity of Operationalization Methods

The *diversity* of FOO's methods must be evaluated subjectively by inspection. These methods can be viewed as experimental vehicles: encoding each new method in FOO tested the *adequacy of the transformational paradigm.* The representational scope of this paradigm is illustrated by the diverse methods applied to the problem of deciding whether an opponent is void:

Inference from observed behavior (§2.6.2)
Recollection of previous conditions (§2.4.2)
Probabilistic analysis (§2.3)
Functional approximation (§2.8.2).

Some additions to the paradigm, including schemas (§3.2) and global variables (§5.4), were needed in order to accommodate some of the methods. By and large, however, the diversity of the encoded methods supports the adequacy of the paradigm as a problem space for operationalization, and suggests that many other methods could be encoded in FOO without significantly altering this paradigm.

9.3.3. Generality of Operationalization Methods

The *generality* of FOO's methods must primarily be evaluated by inspecting their representation as rules expressed in domain-independent terms, since it was not feasible within the scope of the project to test them on a large number of problems drawn from a wide variety of task domains. Except for a few "fact rules" used to simulate unimplemented mechanisms, FOO clearly separates:

General knowledge about operationalization and analysis -- expressed as transformation rules
Specific knowledge about task domains -- expressed mostly as concept definitions

Inheritance knowledge -- expressed as is-a hierarchy, used to apply general rules to specific concepts

In a few cases, generality was explicitly demonstrated by applying the same operationalization strategy to apparently dissimilar problems. For example, the set depletion method (§2.5) was used to operationalize both "get void" and "flush the Queen." Similarly, both the Hearts advice "avoid taking points" and the music task "compose a cantus firmus" were operationalized as heuristic search problems, using substantially the same rules for formulating and refining a heuristic search (§3.6).

Unlike the operationalization strategies, many of FOO's analysis rules were used more than once, and several proved useful in both task domains. Appendix B.2 shows the skewed distribution of rule use in the derivations: most rules were used only once, many were used more often, and a few were used many times.

9.3.4. Mechanical Applicability of Operationalization Methods

The *mechanical applicability* of the operationalization strategies encoded in FOO was demonstrated by mechanically applying each one to one or more problems. This required encoding *analysis methods* (§5) and *domain knowledge* (§6), which in fact comprise most of FOO. The amount of analysis and domain knowledge required to implement general operationalization strategies is discussed further below.

9.3.4.1 Operationality of Operationalization Methods

The concept of *operationality* can be applied to operationalization methods themselves. Operationalizing a piece of advice like "avoid taking points" means figuring out how to apply it to actual task situations, *i.e.*, choosing a card to play. Similarly, operationalizing a general operationalization method means figuring out how to apply it to actual operationalization problems. For example, consider the following strategy (§2.6):

Find an evaluable necessary condition for an unevaluable predicate P.

Formalizing this strategy as "solve P => Q for Q" is not enough to make it operational; applying it requires the ability to *solve* such an equation, for example by elaboration, restructuring, partial matching, and simplification (§2.6.4.1).

Just as the operationality of a piece of advice depends on the capabilities of the runtime mechanism, the operationality of a general operationalization method depends on the analytic power of the operationalization engine. In particular, this engine must be able to

1. Map a specific operationalization problem onto the general method.
2. Test any conditions required by the method.
3. Implement any decisions made by the method.

The more general the method, the more apparatus seems to be needed to make it operational. Perhaps this explains why so much of FOO consists of apparatus supporting a rather small number of operationalization methods. *The contribution of the dissertation lies not so much in the specific set of methods implemented, as in the demonstration of how such methods can be represented and mechanically applied, and in the investigation of the apparatus needed to support them.*

9.3.5. Towards the Automation of AI

The original goal of this research was to encode knowledge about AI in a form suitable for mechanical application. What methods are used in AI? What reasoning is required to apply them to a given problem? How can general methods use knowledge about a specific task domain?

This *scientific* goal stands on its own, independent of the *knowledge engineering* goal of building a machine-aided heuristic programming system. Even if such a system were infeasible, these questions would still deserve attention and the answers provided by the dissertation are a key part of its contribution.

Unsurprisingly, the goal turned out to be quite hard. FOO encodes only a few operationalization methods, some of which appear almost trivially simple, *e.g.*, "to empty a set, remove its elements one by one" (§2.5). The most sophisticated method represented is heuristic search (§3). Most of FOO's rules are included to support such methods, by helping to:

1. Map a problem to a method.
2. Solve subproblems generated by the method.
3. Transform method results into a usable form.

This appears to be an inherent feature of building intelligent programs: some general AI methods are used, but most of the work consists of figuring out how to fit problems into the right form and implement decisions dictated by the methods. This requires both knowledge about the task domain and a substantial amount of general analytical reasoning. Although other, more empirical approaches might lead to a different view, the derivations generated using FOO suggest that

Reformulation is the central process in operationalization.

While machine understanding of AI is an ambitious goal, the progress reported in this dissertation suggests that it is both feasible and worthwhile. Translation of heuristic knowledge into task performance is already a bottleneck in the development of intelligent programs. As these programs grow in size and complexity, the importance of mechanical assistance in their construction can only increase.

Appendix A
Task Descriptions

This appendix describes the Hearts and music tasks used as vehicles for exploring operationalization issues with FOO.

A.1. Rules of Hearts

(The following description of the rules of Hearts is adapted from [Balzer 66].)

1. The game is three-handed, seventeen cards being dealt to each player and one card, the hole card, being placed face down in the center of the table.
2. The person to the left of the dealer leads and, if possible, each player must follow suit. If he has no cards in the suit led ("is void"), he may discard any card from his hand ("break suit"). The person playing the highest card of the suit led wins the trick and leads for the next trick.
3. The person taking in the first trick gets the hole card, which he alone gets to see. He must state whether or not the hole card is a heart.
4. Hearts may not be led until they have been discarded, the leader has nothing else in his hand, or the hole card is a heart.
5. The score is evaluated as follows:
    a. One point to the bad for each heart taken on any trick.
    b. Thirteen points to the bad if the Queen of spades is taken on any trick.
    c. Ten points to the good if the Jack of diamonds is taken.
    d. If the Queen of spades and all 13 hearts are taken by one player, each of the other players gets twenty-six points to the bad. This is termed "shooting the moon."
6. Before the first lead, each player passes four cards to the person on his left, these cards thus becoming part of the next person's hand.
7. The game ends as soon as any player "goes out" by accumulating 100 points (or some other score agreed on before the game). The player with the lowest score wins the game.

FOO's knowledge about the rules of Hearts is encoded in the definitions of TRICK and LEGAL (§C.3).

Derivations DERIV2 (§D.2) through DERIV12 (§D.12) are based on the following Hearts advice:

Avoid taking points (§D.2) (§D.6).

If the Queen of spades is out, flush it (§D.3) without taking it yourself (§D.5).

Don't lead a high card in a suit where an opponent is void.

Get the lead (§D.7).

Get void (§D.8).

Operationalizing these goals generates some evaluation problems:

Decide if the Queen is out (§D.4).

Decide if an opponent is void (§D.9) (§D.11) (§D.12).

Count the number of cards out in a suit (§D.10).

A.2. Cantus Firmus

A *cantus firmus* is a sequence of whole notes satisfying certain musical constraints. DERIV13 and DERIV14 operationalize some of these constraints for a left-to-right tone sequence generator, *i.e.*, as tests that depend only on the notes generated so far and the note about to be generated.

These constraints were selected from a couple of dozen given in a music composition textbook [Salzer 69], which Meehan incorporated in his cantus generation program [Meehan 72].

The constraints used in FOO are shown below along with the concepts in terms of which they are expressed:

C1. A cantus is usually between 8 and 16 notes long.

```
(IN (# CANTUS) (LB:UB 8 16))
```

C2. Dissonant leaps, chromatic half steps, and intervals greater than an octave are not allowed.

```
(FORALL Y (INTERVALS-OF CANTUS) (USABLE-INTERVAL! Y))]

(INTERVALS-OF S) = (EACH I (LB:UB 1 [- (# S) 1])
                         (INTERVAL [S I] [S (+ I 1)]))

(USABLE-INTERVAL! Y) = (NOT (OR [CHROMATIC-HALF-STEP! Y]
                                [DISSONANT-LEAP! Y]
                                [OCTAVE+! Y]))

(CHROMATIC-HALF-STEP! Y) = (= (SIZE Y) (MINOR 2))
```

```
(LEAP! Y) = (> (SIZE Y) (MAJOR 2))

(DISSONANT-LEAP! Y) = (OR [SEVENTH! Y] [AUGMENTED! Y] [DIMINISHED! Y])

(FILTER (INTERVALS) USABLE-INTERVAL!) = (SET m2 M2 m3 M3 P4 P5 m6 M6 P8)
```

C3. The range of the cantus should not exceed a major tenth.

```
(=< (MELODIC-RANGE CANTUS) (MAJOR 10))

(MELODIC-RANGE S) = (SIZE (INTERVAL (LOWEST S) (HIGHEST S)))
```

C4. The climax note of the cantus should not be repeated.

```
(= (#OCCURRENCES (CLIMAX CANTUS) CANTUS) 1)

(CLIMAX S) = (HIGHEST S)

(#OCCURRENCES N S) = (# (SET-OF X S (= X N)))
```

The informality in the last definition arises from quantifying over the sequence S as though it were a set (§2.11.1.1).

Appendix B
Transformation Rules

This appendix places FOO's rules in a taxonomy (§B.1), shows how often different rules are used in the derivations (§B.2), describes how FOO interprets transformation rules (§B.3), and lists them in a precise machine-generated notation (§B.4).

B.1. Rule Taxonomy

The following taxonomy of FOO's rules for the most part follows the classifications used in the text. Some rules appear under more than one classification. The top-level categories are:

1. Analysis rules (§5)
2. Control rules for subgoaling and global communication (§5.4)
3. Domain-specific facts (§1.3.3)
4. Operationalization strategies (§2)
5. Translation rules (§5.8)

The rules are shown in their entirety in Appendix B.4; the abbreviated versions in parentheses are intended to suggest the nature of each rule by showing the flattened list of literal (non-variable) symbols contained in its left- and right-hand side, without regard to list structure.

```
RECORD FILE DSK: (RLT319 . QZG) OPENED 20-MAR-81 18:22:45
NIL
<73>p-f any-rule

 ANY-RULE:
      ANALYSIS-RULE:
           ACTION-RULE:  RULE254 (CHANGE -> ACTION)
                         RULE339 (= CHOOSE -> = SET)
                         RULE353 (NOT = -> = ME)
                         RULE358 (CAUSE AT -> MOVE)
                         RULE368 (UNDO AT -> MOVE)
           EXAMPLE-RULE:  RULE84 (FIND-ELT -> $CONSTANT!)
                          RULE109 (NOT AND FORALL -> => AND NOT FIND-ELT)
                          RULE142 (PREDICATE FIND-ELT -> AND IN PREDICATE)
                          RULE347 (EXISTS $CONSTANT! -> PREDICATE $CONSTANT!)
                          RULE364 (IN $UNBOUND-VAR! PROJECT -> T)
           FUNCTIONAL-DOMAIN-RULE:  RULE195 (DOMAIN -> DOMAIN)
```

```
                         RULE196 (DOMAIN $UNARY! -> DOMAIN)
                         RULE197 (DOMAIN SUIT-OF -> CARDS)
FUNCTIONAL-RECURRENCE-RULE:  RULE211 (= $UNBOUND-VAR! -> = $UNBOUND-VAR! LAMBDA)
                             RULE212 (= $UNBOUND-VAR! -> = $UNBOUND-VAR! INVERSE LAMBDA)
                             RULE315 (REFORMULATE -> LAMBDA)
FUNCTIONAL-RULE:
     DEFUNCTIONALIZE-RULE:  RULE273 ($UNAPPLIED! -> ?)
     FUNCTIONALIZE-RULE:  RULE297 (BOOLEAN-OP -> BOOLEAN-OP-$FUNCTIONAL!)
                          RULE298 (NOT -> NOT-$FUNCTIONAL!)
                          RULE305 (ANY-CONCEPT -> ANY-CONCEPT)
                          RULE393 (LAMBDA FUNCTION -> FUNCTION LAMBDA)
INTERSECTION-SEARCH-RULE:  RULE57 (DURING -> NIL)
                           RULE199 (COMMON-SUPERSET SET-OF SET-OF -> ?)
                           RULE308 (CHOICE-SEQ-OF -> NIL)
PARTIAL-MATCH-RULE:  RULE30 (TRANSITIVE $SET! -> SUBSET $SET!)
                     RULE43 (REFLEXIVE -> AND =)
                     RULE91 (REFLEXIVE -> AND =)
                     RULE172 (IN PROJECT -> IN)
                     RULE192 (= QUANTIFIER-PREDICATE QUANTIFIER-PREDICATE -> =)
                     RULE322 (=> EXISTS -> AND IN =)
                     RULE379 (=> FORALL FORALL -> SUBSET)
                     RULE380 (= FORALL FORALL -> = OR IN DIFF)
                     RULE388 (SUBSET SET-OF SET-OF -> SUBSET)
PROBLEM-MERGING-RULE:  RULE297 (BOOLEAN-OP -> BOOLEAN-OP-$FUNCTIONAL!)
                       RULE360 (AND ACHIEVE -> ACHIEVE AND)
PROBLEM-REDUCTION-RULE:
     HEURISTIC-EQUIVALENCE-PRESERVING-RULE:  RULE213 (LAMBDA -> ?)
                                             RULE215 (LAMBDA -> ?)
                                             RULE290 (NUMERICAL-PREDICATE + $CONSTANT!
                                                       $CONSTANT! -> NUMERICAL-PREDICATE
                                                       $CONSTANT!)
                                             RULE374 (BEFORE BOOLEAN-$CONSTANT! ->
                                                              BOOLEAN-$CONSTANT!)
     REDUCE->NEC-COND-RULE:  RULE350 (ACHIEVE AND -> ACHIEVE AND)
                             RULE354 (AND = -> AND)
                             RULE369 (IN SET-OF -> ?)
     REDUCE->NON-EQUIVALENT-RULE:  RULE238 (CHOOSE -> ?)
                                   RULE366 (ACHIEVE PREVIOUS -> ACHIEVE)
     REDUCE->SUFF-COND-RULE:  RULE193 (= IN $VAR! -> = SET-OF $VAR! DOMAIN $VAR!)
                              RULE224 (=> AND -> =>)
                              RULE225 (=> OR -> AND NOT OR =>)
                              RULE226 (=> => -> AND =>)
                              RULE277 (CHANGE EXTREME -> COMPARATIVE EXTREME CHANGE
                                               EXTREME)
     SPECIAL-CASE-REDUCTION-RULE:  RULE19 (DIFF JOIN -> JOIN)
                                   RULE300 (NEXT EXTREME -> EXTREME CHANGE)
                                   RULE303 (SET-OF -> NIL)
                                   RULE324 (= EXTREME EXTREME -> IN EXTREME)
                                   RULE325 (= -> IN INDICES-OF)
                                   RULE362 (QUANTIFIER-PREDICATE IN ->
                                            QUANTIFIER-PREDICATE)
          CHECK-NEC-COND-RULE:  RULE304 (IN -> NOT COMPARATIVE EXTREME)
          CHECK-SUFF-COND-RULE:  RULE336 (=> $DETECTOR! $DETECTOR! -> SUBSET)
                                 RULE383 (=> TRANSITIVE TRANSITIVE -> TRANSITIVE)
PROBLEM-SEPARATION-RULE:
     INTRODUCE-SPLIT-RULE:  RULE163 (RANGE LOC -> PARTITION HANDS PLAYERS PILES PLAYERS
                                           SET DECK POT HOLE)
                            RULE164 (DIFF DIFF -> UNION DIFF INTERSECT)
                            RULE165 (INTERSECT SET -> IF IN SET NIL)
                            RULE256 (PREDICATE -> AND = PREDICATE)
                            RULE258 (PREDICATE -> AND => NOT CAUSE => NOT UNDO)
                            RULE274 (PREFIX + 1 -> APPEND PREFIX LIST NTH + 1)
                            RULE276 (PREDICATE NEXT -> OR AND NOT CHANGE PREDICATE AND
                                                CHANGE PREDICATE)
                            RULE338
                            RULE340 (= -> AND $ORDERING! $ORDERING!)
                            RULE344 (UNDO = -> AND = BECOME)
                            RULE366 (AT -> OR WAS-DURING CURRENT ACTION CAUSE AT BEFORE
                                        CURRENT ACTION AT)
                            RULE370 (# SET-OF -> - # SET-OF BEFORE CURRENT # SET-OF
                                       WAS-DURING CURRENT UNDO)
                            RULE385 (SET-OF OR -> IF SET-OF OR)
```

```
                              RULE389
      PROPAGATE-SPLIT-RULE:   RULE7 (REMOVE-1-FROM SET-OF -> UNDO AND IN)
                              RULE10 (IN SET-OF -> AND IN)
                              RULE17 (SUBSET SET -> AND IN)
                              RULE35 (AFFECT AND -> AND AFFECT)
                              RULE59 (EXISTS AND = -> AND IN)
                              RULE64 (= THE -> AND IN)
                              RULE121 (SET-OF COLLECTION -> UNION IF SET NIL)
                              RULE131 (= THE -> AND IN)
                              RULE149 (IN $FILTER! -> AND PREDICATE IN)
                              RULE182 (IN SET -> OR =)
                              RULE284 (HOMOMORPHISM JOIN -> JOIN HOMOMORPHISM)
                              RULE287 (IF -> IF)
                              RULE321 (=> AND -> AND =>)
                              RULE361 (IN FILTER -> AND IN)
                              RULE371 (BEFORE -> BEFORE)
                              RULE375 (# SET-OF NOT -> - # # SET-OF)
                              RULE393 (LAMBDA FUNCTION -> FUNCTION LAMBDA)
  RECURSIVE-DEFINITION-RULE:
      CHOICE-SEQ-RULE:   RULE307 (CHOICE-SEQ-OF EACH -> APPLY APPEND EACH CHOICE-SEQ-OF)
                         RULE308 (CHOICE-SEQ-OF -> NIL)
                         RULE309 (CHOICE-SEQ-OF CHOOSE -> LIST CHOOSE)
      DEPENDENCE-RULE:   RULE203 (DEPENDENCE $ATOM! -> D+ D* DEPENDENCE $ATOM! DEPENDENCE)
                         RULE204 (DEPENDENCE $ATOM! -> NIL)
                         RULE205 (DEPENDENCE INCR1-DECR2 -> D+ DEPENDENCE D- DEPENDENCE)
                         RULE207 (DEPENDENCE INCR1 $ATOM! -> DEPENDENCE $ATOM!)
                         RULE208 (DEPENDENCE -> INCREASING)
  RESTRUCTURE-RULE:
      TRANSFER-RULE:   RULE13 (QUANTIFIER-PREDICATE INJECTIVE-MAPPED-QUANTIFIER ->
                                 QUANTIFIER-PREDICATE)
                       RULE135 (EXISTENTIAL-QUANTIFIER SET-OF ->
                                 EXISTENTIAL-QUANTIFIER AND)
                       RULE161 (QUANTIFIER -> QUANTIFIER PROJECT)
                       RULE187 (UNION SET SET -> UNION SET)
                       RULE219 (FORALL SET-OF -> FORALL =>)
                       RULE326 (IN -> IN INDICES-OF)
                       RULE328 (QUANTIFIER-PREDICATE -> QUANTIFIER-PREDICATE INDICES-OF)
                       RULE330 (FORALL INDICES-OF IN -> FORALL INDICES-OF NOT AFTER)
      TRANSPOSE-RULE:
          PURE-TRANSPOSE-RULE:   RULE5 (DIFF SET-OF -> SET-OF DIFF)
                                 RULE35 (AFFECT AND -> AND AFFECT)
                                 RULE108 (QUANTIFIER-PREDICATE AND ->
                                           AND QUANTIFIER-PREDICATE AND)
                                 RULE170 (PROJECT DIFF -> DIFF PROJECT PROJECT)
                                 RULE220 (=> QUANTIFIER -> QUANTIFIER =>)
                                 RULE278 (CHANGE $PROJECTION! -> $PROJECTION! CHANGE)
                                 RULE280 ($PROJECTION! COLLECTION -> COLLECTION)
                                 RULE283 (NEXT $FUNCTIONAL! -> NEXT)
                                 RULE296 (AND NOT NOT -> AND NOT OR)
                                 RULE321 (=> AND -> AND =>)
                                 RULE329 (PREDICATE QUANTIFIER-PREDICATE ->
                                           QUANTIFIER-PREDICATE PREDICATE)
                                 RULE359 (AND UNTIL -> UNTIL AND)
                                 RULE360 (AND ACHIEVE -> ACHIEVE AND)
                                 RULE371 (BEFORE -> BEFORE)
          QUASI-TRANSPOSE-RULE:   RULE58 (DURING EACH -> EXISTS DURING)
                                  RULE100 (DURING FOR-SOME -> EXISTS DURING)
                                  RULE171 (PROJECT SET -> SET)
                                  RULE284 (HOMOMORPHISM JOIN -> JOIN HOMOMORPHISM)
  SIMPLIFY-RULE:
      OPPORTUNISTIC-EVALUATION-RULE:   RULE47 (=> T -> T)
                                       RULE179 (REFLEXIVE -> T)
                                       RULE184 (OR T -> T)
                                       RULE288 (# COLLECTION -> NON-NEGATIVE-INTEGER)
                                       RULE289 (# NIL -> 0)
                                       RULE293 (IN ONE-OF -> T)
                                       RULE337 (SUBSET PREFIX -> T)
      SIMPLIFY->NON-CONSTANT-RULE:   RULE11 (AND T -> AND)
                                     RULE12 (=> T -> ?)
                                     RULE88 (NOT NOT -> ?)
                                     RULE156 (QUANTIFIER-PREDICATE -> ?)
                                     RULE178 (JOIN NIL -> JOIN)
```

```
                                          RULE177 (COMM-CONN COMM-CONN -> COMM-CONN)
                                          RULE178 (COMM-CONN -> ?)
                                          RULE185 (IF T -> ?)
                                          RULE188 (COMM-CONN -> COMM-CONN)
                                          RULE216 (INVERSE -> ?)
                                          RULE253 (LB:UB -> SET)
                                          RULE255 (QUANTIFIER-PREDICATE COLLECTION -> ?)
                                          RULE281 (ONE-OF COLLECTION -> ?)
                                          RULE310 (APPLY APPEND EACH LIST -> EACH)
                                          RULE341 (COMM-CONN $IDENTITY! -> ?)
                                          RULE343 (COMM-CONN $IDENTITY! -> ?)
          SYSTEMATIC-EVALUATION-RULE:  RULE236 (COMPUTABLE $CONSTANTL! -> $CONSTANT!)
CONTROL-RULE:  RULE34
               RULE52
               RULE53
               RULE190
               RULE268
               RULE269
               RULE302
               RULE312
               RULE313
               RULE320
               RULE331 (THEN -> ?)
               RULE334
     MAKE-ASSUMPTION-RULE:  RULE15 (PARTITION -> UNION)
                            RULE32
                            RULE157 (=> -> ?)
                            RULE221
                            RULE342 (= $CONSTANT! -> T)
                            RULE349 (=> -> T)
     USE-ASSUMPTION-RULE:  RULE346 (ANY-CONCEPT -> ANY-CONCEPT)
                           RULE351 (ACHIEVE -> ACHIEVE AND =)
                           RULE354 (AND = -> AND)
                           RULE364 (IN $UNBOUND-VAR! PROJECT -> T)
          ALL-PURPOSE-RULE:  RULE301 (ANY-CONCEPT -> FILL-IN)
     VARIABLE-RULE:  RULE127 ($BOUND-VAR! -> ?)
                     RULE194 (= $UNBOUND-VAR! -> T)
                     RULE271 (= $UNBOUND-VAR! -> T)
FACT-RULE:  RULE1 (HAS OWNER-OF CARD CARD -> T)
            RULE2 (SUIT-OF QS -> S)
            RULE61 (ACTIONS-OF -> PLAY-CARD)
            RULE62 (IN QS CARDS -> T)
            RULE146 (= PLAYER-OF CARD -> = CARD-OF CARD)
            RULE163 (RANGE LOC -> PARTITION HANDS PLAYERS PILES PLAYERS SET DECK POT HOLE)
            RULE173 (IN ME PLAYERS -> T)
            RULE180 (AT CARD DECK -> NIL)
            RULE197 (DOMAIN SUIT-OF -> CARDS)
            RULE228 (LEGAL CARD -> = CARD CARD-OF)
            RULE279 (CHANGE CANTUS-1 -> LIST NEXT NOTE)
            RULE285 (NEXT CANTUS-1 -> APPEND CANTUS-1 LIST NEXT NOTE)
            RULE299 (NOT OR HIGHER = -> LOWER)
            RULE357 (BEFORE CURRENT ROUND-IN-PROGRESS AT CARD PILE -> NIL)
            RULE363 (SUBSET CARDS-PLAYED CARDS -> T)
            RULE365 (IN LEADER PLAYERS -> T)
            RULE372 (BEFORE CURRENT ROUND-IN-PROGRESS IN-POT CARD -> NIL)
            RULE373 (AT CARD HOLE -> NIL)
            RULE376 (# CARDS-IN-SUIT -> 13)
OPERATIONALIZATION-STRATEGY:
     APPROXIMATION-RULE:  RULE154 (COMPARATIVE -> $UNARY-PREDICATE!)
          FUNCTIONAL-DEPENDENCE-RULE:  RULE202 (EVAL FORMULA ->
                                                FUNCTION-OF DEPENDENCE FORMULA)
                                       RULE210 (FUNCTION-OF DEPENDENCE -> FUNCTION-OF
                                                                 INCREASING)
     CHOICE-RULE:
          CONSTRAIN-CHOICE-RULE:  RULE156 (ACHIEVE PREDICATE -> CHOOSE SET-OF PREDICATE)
          DESIGN-CHOICE-RULE:  RULE256 (PREDICATE -> AND = PREDICATE)
     DISJOINT-SUBSETS-RULE:  RULE189 (PREDICATE $SET!-$UNKNOWN! -> DISJOINT $SET!-$UNKNOWN!)
                             RULE198 (DISJOINT $UNKNOWN! -> PR-DISJOINT $UNKNOWN!
                                                   COMMON-SUPERSET $UNKNOWN!)
                             RULE200 (PR-DISJOINT $UNKNOWN! -> PR-DISJOINT $UNKNOWN! FILTER
                                                       $UNARY! FILTER $UNARY!)
     HISTORY-RULE:  RULE234 (PREDICATE -> WAS-DURING CURRENT PREDICATE)
```

```
                    RULE356 (AT -> OR WAS-DURING CURRENT ACTION CAUSE AT BEFORE CURRENT ACTION
                            AT)
                    RULE370 (# SET-OF -> - # SET-OF BEFORE CURRENT # SET-OF WAS-DURING CURRENT
                            UNDO)
        HSM-RULE:
            HSM-INSTANTIATE:  (RULE311 RULE314 RULE316 RULE317)
            HSM-PROBLEM-STATEMENT:  RULE306
            HSM-REFINE:
                HSM-CACHE:  RULE333
                HSM-FIX-UP:  RULE318 (LAMBDA SET-OF
                                  PREDICATE -> LAMBDA SET-OF POSSIBLE PREDICATE)
                MOVE-CONSTRAINT:
                    GENERATOR->PRECOMPUTE:  RULE386
                    PATH->STEP-CONSTRAINT:
                        PATH-ORDER->STEP-ORDER:  RULE338
                        PATH-TEST->STEP-TEST:  (RULE327 RULE389)
                        SIMPLIFY-BY-INDUCTION:  RULE390
                    SOLUTION->PATH-CONSTRAINT:
                        SOLUTION-TEST->PATH-ORDER:  RULE335
                        SOLUTION-TEST->PATH-TEST:  RULE323
                    SOLUTION-TEST->COMPLETION-TEST:  RULE378
                    STEP-TEST->GENERATOR:  RULE384
                REDUCE-SEARCH-DEPTH:  RULE332
        NECESSARY-CONDITION-RULE:  RULE319 (PREDICATE -> => PREDICATE)
        PIGEON-HOLE-RULE:  RULE169 (IN -> NOT IN DIFF RANGE)
        SET-DEPLETION-RULE:  RULE6 (ACHIEVE EMPTY -> UNTIL ACHIEVE REMOVE-1-FROM)
        SUFFICIENT-CONDITION-RULE:  RULE227 (EVAL PREDICATE -> EVAL => PREDICATE)
        TEMPORAL-GOAL-RULE:  RULE258 (PREDICATE -> AND => NOT CAUSE => NOT UNDO)
                             RULE266 (PREDICATE SEQUENCE -> FORALL LB:UB 0 - # SEQUENCE 1
                                               PREDICATE LAMBDA PREFIX SEQUENCE)
                             RULE276 (PREDICATE NEXT -> OR AND NOT CHANGE PREDICATE AND CHANGE
                                               PREDICATE)
                             RULE282 (ACHIEVE -> NEXT)
                             RULE300 (NEXT EXTREME -> EXTREME CHANGE)
                             RULE366 (ACHIEVE PREVIOUS -> ACHIEVE)
    TRANSLATE-RULE:
        ELABORATE-RULE:  RULE4 (SUBSET -> EMPTY DIFF)
                         RULE14 (IN PROJECT -> EXISTS =)
                         RULE15 (PARTITION -> UNION)
                         RULE124 ($EXPANDABLE! -> ?)
                         RULE381 (INDICES-OF -> LB:UB 1 #)
        LATERAL-REPHRASE-RULE:  RULE153 (COMPARATIVE -> COMPARATIVE)
                                RULE159 (NOT EXISTS -> EMPTY SET-OF)
                                RULE218 (= FILTER -> => IN)
        RECOGNIZE-RULE:  RULE123 ($RECOGNIZABLE! -> ?)
                         RULE162 (EXISTS = -> IN)
                         RULE267 (LAMBDA -> ?)
                         RULE291 (= # SET-OF 0 -> NOT EXISTS)
                         RULE292 (EXISTS = -> IN)
                         RULE294 (IF NIL -> AND NOT)
                         RULE352 (= NIL -> NOT)
                         RULE358 (CAUSE AT -> MOVE)
                         RULE367 (NOT -> ?)
                         RULE368 (UNDO AT -> MOVE)
                         RULE377 (UNDO NOT -> CAUSE)
                         RULE382 (DIFF LB:UB LB:UB -> LB:UB - 1)
                         RULE387 (>= # SET-OF 1 -> EXISTS)
                         RULE391 (IF T -> OR NOT)
                         RULE392 (=< $NON-NEGATIVE! 0 -> = $NON-NEGATIVE! 0)
NIL
<74>recordfile

RECORD FILE DSK: (RLT319 . QZG) CLOSED 20-MAR-81 18:26:46
```

B.2. Rule Usage

The pattern of rule usage is indicated by the histogram below. The most frequently used rules were RULE124, which plugs in the definition of a concept, and its inverse, RULE123, which substitutes the name of a concept for an expression that matches its definition. Each rule is classified according to a category in the rule taxonomy shown in Appendix B.1. Many of the 41 rules used in both the Hearts and music tasks are control rules.

```
RECORD FILE DSK: (DAT319 . QZG) OPENED 20-MAR-81 03:56:21
NIL
<1>p-data

230 rules used in 714 steps in 13 derivations
Average # applications of rules used in derivations = 3
Histogram of rule use in derivations:
N    Rules used N times  (+ flags rules used in both Hearts and music domains)
119: 1 rules  ((+ RULE124 ELABORATE-RULE))
42:  1 rules  ((+ RULE123 RECOGNIZE-RULE))
20:  2 rules  ((+ RULE268 CONTROL-RULE) (+ RULE269 CONTROL-RULE))
18:  2 rules  ((+ RULE178 SIMPLIFY->NON-CONSTANT-RULE)
               (+ RULE236 SYSTEMATIC-EVALUATION-RULE))
15:  1 rules  ((+ RULE127 VARIABLE-RULE))
14:  4 rules  ((+ RULE34 CONTROL-RULE)
               (+ RULE301 ALL-PURPOSE-RULE)
               (+ RULE313 CONTROL-RULE)
               (+ RULE341 SIMPLIFY->NON-CONSTANT-RULE))
12:  2 rules  ((+ RULE53 CONTROL-RULE)
               (+ RULE177 SIMPLIFY->NON-CONSTANT-RULE))
9:   2 rules  ((+ RULE284 QUASI-TRANSPOSE-RULE PROPAGATE-SPLIT-RULE)
               (+ RULE346 USE-ASSUMPTION-RULE))
8:   2 rules  ((RULE11 SIMPLIFY->NON-CONSTANT-RULE)
               (+ RULE343 SIMPLIFY->NON-CONSTANT-RULE))
7:   2 rules  ((+ RULE194 VARIABLE-RULE) (+ RULE271 VARIABLE-RULE))
6:   2 rules  ((RULE43 MATCH-RULE) (+ RULE179 OPPORTUNISTIC-EVALUATION-RULE))
5.   7 rules  ((+ RULE12 SIMPLIFY->NON-CONSTANT-RULE)
               (RULE35 PURE-TRANSPOSE-RULE PROPAGATE-SPLIT-RULE)
               (RULE176 SIMPLIFY->NON-CONSTANT-RULE)
               (+ RULE312 CONTROL-RULE)
               (+ RULE323 SOLUTION-TEST->PATH-TEST)
               (RULE329 PURE-TRANSPOSE-RULE)
               (+ RULE334 CONTROL-RULE))
4:   6 rules  ((RULE2 FACT-RULE)
               (RULE10 PROPAGATE-SPLIT-RULE)
               (RULE57 INTERSECTION-SEARCH-RULE)
               (+ RULE153 LATERAL-REPHRASE-RULE)
               (RULE173 FACT-RULE)
               (RULE292 RECOGNIZE-RULE))
3:   15 rules ((RULE30 MATCH-RULE)
               (RULE32 MAKE-ASSUMPTION-RULE)
               (RULE64 PROPAGATE-SPLIT-RULE)
               (RULE88 SIMPLIFY->NON-CONSTANT-RULE)
               (RULE91 MATCH-RULE)
               (RULE135 TRANSFER-RULE)
               (RULE151 REDUCE-SUFF-COND-RULE)
               (RULE172 MATCH-RULE)
               (RULE184 OPPORTUNISTIC-EVALUATION-RULE)
               (+ RULE192 MATCH-RULE)
               (RULE267 RECOGNIZE-RULE)
               (RULE291 RECOGNIZE-RULE)
               (+ RULE327 PATH-TEST->STEP-TEST)
               (+ RULE331 CONTROL-RULE)
               (RULE358 RECOGNIZE-RULE ACTION-RULE))
2:   44 rules ((RULE1 FACT-RULE)
               (RULE6 SET-DEPLETION-RULE)
               (RULE7 PROPAGATE-SPLIT-RULE)
```

```
                    (RULE47 OPPORTUNISTIC EVALUATION-RULE)
                    (RULE58 QUASI-TRANSPOSE-RULE)
                    (RULE59 PROPAGATE-SPLIT-RULE)
                    (RULE109 EXAMPLE-RULE)
                    (RULE121 PROPAGATE-SPLIT-RULE)
                    (+ RULE131 PROPAGATE-SPLIT-RULE)
                    (RULE139 PROPAGATE-SPLIT-RULE)
                    (RULE154 APPROXIMATION-RULE)
                    (RULE155 SIMPLIFY->NON-CONSTANT-RULE)
                    (RULE162 RECOGNIZE-RULE)
                    (+ RULE182 PROPAGATE-SPLIT-RULE)
                    (+ RULE185 SIMPLIFY->NON-CONSTANT-RULE)
                    (+ RULE188 SIMPLIFY->NON-CONSTANT-RULE)
                    (RULE204 DEPENDENCE-RULE)
                    (RULE205 DEPENDENCE-RULE)
                    (RULE279 FACT-RULE)
                    (RULE281 SIMPLIFY->NON-CONSTANT-RULE)
                    (RULE283 PURE-TRANSPOSE-RULE)
                    (RULE287 PROPAGATE-SPLIT-RULE)
                    (RULE288 OPPORTUNISTIC-EVALUATION-RULE)
                    (RULE290 HEURISTIC-EQUIVALENCE-PRESERVING-RULE)
                    (RULE302 CONTROL-RULE)
                    (+ RULE306 DETECT-APPLICABILITY HSM-INSTANTIATE)
                    (+ RULE307 CHOICE-SEQ-RULE)
                    (+ RULE309 CHOICE-SEQ-RULE)
                    (+ RULE310 SIMPLIFY->NON-CONSTANT-RULE)
                    (+ RULE311 HSM-INSTANTIATE)
                    (+ RULE314 HSM-INSTANTIATE)
                    (+ RULE315 FUNCTIONAL-RECURRENCE-RULE)
                    (+ RULE317 HSM-INSTANTIATE)
                    (RULE319 NECESSARY-CONDITION-RULE)
                    (RULE320 CONTROL-RULE)
                    (+ RULE323 TRANSFER-RULE)
                    (+ RULE340 INTRODUCE-SPLIT-RULE)
                    (RULE354 USE-ASSUMPTION-RULE REDUCE->NEC-COND-RULE)
                    (RULE361 PROPAGATE-SPLIT-RULE)
                    (RULE368 RECOGNIZE-RULE ACTION-RULE)
                    (RULE371 PURE-TRANSPOSE-RULE PROPAGATE-SPLIT-RULE)
                    (RULE379 MATCH-RULE)
                    (RULE383 CHECK-SUFF-COND-RULE)
                    (RULE393 PROPAGATE-SPLIT-RULE FUNCTIONALIZE-RULE))
1:    130 rules
0.    7 rules  ((RULE156 CONSTRAIN-CHOICE-RULE)
                    (RULE211 FUNCTIONAL-RECURRENCE-RULE)
                    (RULE258 INTRODUCE-SPLIT-RULE TEMPORAL-GOAL-RULE)
                    (RULE278 PURE-TRANSPOSE-RULE)
                    (RULE280 PURE-TRANSPOSE-RULE)
                    (RULE285 FACT-RULE)
                    (RULE303 SPECIAL-CASE-REDUCTION-RULE))
NIL
<2>recordfile

RECORD FILE DSK: (DAT319 . QZG) CLOSED 20-MAR-81 03:57:09
```

B.3. Interpretation of Rules in FOO

This section uses a precise rule notation to give a sense of concreteness and to illustrate how the rule interpreter works. Appendix B.4 lists all the rules in this notation.

To illustrate, consider the following somewhat informally stated rule:

> RULE284: (f ... (+ e_1 ... e_n) ...) -> (+' (f ... e_1 ...) ... (f ... e_n ...))
> where + is an addition-like operator over the domain of f, +' is the corresponding operator over the range of f, and (lambda (x) (f ... x ...)) is a *homomorphism* with respect to + and +'

A transformation rule in FOO has the following form (braces indicate optional components):

```
RULEnnn:  {** <component>} * <lhs> -> <rhs> {IF <test>}
```

To apply RULEnnn, the <lhs> pattern is matched against an expression. The <component> pattern, if present, is matched against a selected argument of the expression. If the <test> is satisfied, the <rhs> is instantiated and substituted for the expression.

Thus RULE284 is stated in a precise, machine-oriented notation as

```
RULE284:  ** (?S : (JOIN & ?L))
          * (?E : ((?F : HOMOMORPHISM) & ?M)))
          -> ((! $ADD-OF ?F)
              & (! $DISTRIBUTE ?X ?L (!! ! $SUBST ?X ?S ?E)))
```

The symbol JOIN corresponds to the addition-like operator +, and (! $ADD-OF ?F) corresponds to +'.

"Machine-oriented notation" is not the same as "internal representation." A rule is represented internally in FOO as a LISP atom with properties corresponding to various rule components. The precise notation shown above is mechanically generated from this internal representation, with punctuation added for readability.

The notation uses the following symbols:

* precedes pattern to be matched against the expression e to which the rule is applied

** precedes pattern to be matched against a specified component of e

-> precedes pattern to be substituted for e in the current overall expression

IF precedes condition to be satisfied in order for rule to be applied

? prefix denotes a variable, *e.g.*, ?S

: is a binding operator, *e.g.*, `(?F : HOMOMORPHISM)`

$ prefix indicates a procedure, *e.g.*, `$SUBST`

& is a concatenation operator, *e.g.*, `(JOIN & ?L)`

! is an evaluation operator, *e.g.*, `(! $ADD-OF ?F)`

!! is a quoting (evaluation-delaying) operator, *e.g.*, `(!! ! $SUBST ?X ?S ?E)`

`Q*V` generates unique variable names, *e.g.*, `(! Q*V ?X)`

FOO decides if a rule can be applied to an expression by matching the left-hand side of the rule against the expression. The process of matching a rule pattern to an expression is performed top-down without backtracking, and can be described recursively:

1. Match unbound <var> to e by binding <var> to e.
2. Match bound <var> to e by matching its binding to e.
3. Match literal c to e by testing if e is-a c.
4. Match (<pat$_1$> : <pat$_2$>) to e by matching <pat$_1$> to e and <pat$_2$> to e.
5. Match (<pat$_1$> & <pat$_2$>) to (f e$_1$ ... e$_n$) by matching <pat$_1$> to f and <pat$_2$> to (e$_1$ ... e$_n$).
6. Match (<pat$_1$> ... <pat$_n$>) to (e$_1$ ... e$_n$) by matching <pat$_1$> to e$_1$, ..., <pat$_n$> to e$_n$.

The left-hand side of RULE284 is a two-part pattern; one part, prefixed by a single asterisk, is matched against an expression, and the other, prefixed by a double asterisk, is matched against an argument of that expression. This is illustrated by the transformation

```
             (DURING (SCENARIO (EACH P1 (PLAYERS) (PLAY-CARD P1))
                               (TAKE-TRICK (TRICK-WINNER)))
                     (TAKE-POINTS ME))
6:4          --- [DISTRIBUTE by RULE284] --->
             (OR [DURING (EACH P1 (PLAYERS) (PLAY-CARD P1))
                         (TAKE-POINTS ME)]
                 [DURING (TAKE-TRICK (TRICK-WINNER)) (TAKE-POINTS ME)])
```

The pattern `(?E : ((?F : HOMOMORPHISM) & ?M)))` is matched against the expression

```
(DURING (SCENARIO (EACH P1 (PLAYERS) (PLAY-CARD P1))
                  (TAKE-TRICK (TRICK-WINNER)))
        (TAKE-POINTS ME))
```

This is done by searching through an is-a hierarchy (§1.3.2.1) to find a path from `DURING` to `HOMOMORPHISM`. The variable `?E` is bound to the whole expression, `?M` is bound to the list comprising its two arguments, and `?F` is bound to the function `DURING`.

Similarly, the subpart pattern `(?S : (JOIN & ?L))` is matched against the sub-expression

```
     (SCENARIO (EACH P1 (PLAYERS) (PLAY-CARD P1))
               (TAKE-TRICK (TRICK-WINNER)))
```

This is done by by finding a path from SCENARIO to JOIN. The variable ?S is bound to the sub-expression and ?L is bound to the list comprising its two arguments. Note that the user specifies which sub-expression to match the subpart pattern against.

The matching process has two purposes. One is to verify the appropriateness of applying the rule. If a rule does not match an expression, FOO will not attempt to apply it. For example, RULE284 produces a valid transformation provided the homomorphism condition is satisfied. FOO would refuse to apply RULE284 at step 6:4 if either of the relations DURING is-a HOMOMORPHISM and SCENARIO is-a JOIN were not satisfied.

The other purpose of matching is to bind variables used in the pattern on the right-hand side of the rule. This pattern is instantiated by filling in variable bindings and executing any embedded procedures. The instantiation process is illustrated below for the right-hand side of RULE284:

```
     ((! $ADD-OF ?F) & (! $DISTRIBUTE ?X ?L (!! ! $SUBST ?X ?S ?E)))
```

First the sub-pattern (! $ADD-OF ?F), corresponding to the addition-like operator +' in the informal statement of RULE284, is evaluated by filling in DURING as the binding for ?F, and applying the procedure $ADD-OF to DURING. FOO finds the stored relation ADD of DURING is OR in its semantic network of object-attribute-value triples (§1.3.2), and returns the function OR. Thus the expression to be substituted for the original expression will be in the form of a disjunction. Note that the symbol ! in (! $ADD-OF ?F) indicates procedure application; FOO would instantiate the pattern ($ADD-OF ?F) as the literal expression ($ADD-OF DURING).

The procedure $DISTRIBUTE is a mapping quantifier similar to MAPCAR in LISP. The sub-pattern (! $DISTRIBUTE ?X ?L (!! ! $SUBST ?X ?S ?E)) is evaluated by binding the variable ?X to successive elements of the list bound to ?L and instantiating (!! ! $SUBST ?X ?S ?E) in the context of each such binding. This in turn is done by using the procedure $SUBST to substitute the expression bound to ?X for the expression bound to ?S throughout the expression bound to ?E. The symbol !! keeps FOO from trying to fill in the binding of ?X prematurely, *i.e.*, before the procedure $DISTRIBUTE is applied. The result returned by $DISTRIBUTE is the list of expressions produced by instantiating (!! ! $SUBST ?X ?S ?E) for each ?X in ?L.

The ampersand in the right-hand side of RULE284 acts as a concatenation operator: it tells FOO to append this list to the function OR found earlier. This produces the disjunction

```
(OR [DURING (EACH P1 (PLAYERS) (PLAY-CARD P1))
            (TAKE-POINTS ME)]
    [DURING (TAKE-TRICK (TRICK-WINNER)) (TAKE-POINTS ME)])
```

This is substituted at step 6 : 4 for the expression bound to the left-hand side of RULE284.

B.4. List of Rules

FOO's rules are listed below in the notation explained in Appendix B.3.

```
RECORD FILE DSK: (RUL319 . QZG) OPENED 20-MAR-81 18:32:13
NIL
<89>p-rules nil

230 rules in use as of "20-MAR-81 18:32:13"

(RULE1 (FACT-RULE USED 2 TIMES) * (HAS (OWNER-OF ?C) ?C) -> T)
Used in: ((DERIV3 : 22 70))

(RULE2 (FACT-RULE USED 4 TIMES) * (SUIT-OF QS) -> S)
Used in: ((DERIV3 : 25 30 72) (DERIV5 : 45))

(RULE4 (ELABORATE-RULE USED 1 TIMES)
       * (SUBSET ?X ?Y)
       -> (EMPTY (DIFF ?X ?Y)))
Used in: ((DERIV3 : 37))

(RULE5 (PURE-TRANSPOSE-RULE USED 1 TIMES)
       * (DIFF (SET-OF ?X ?S1 ?P) ?S2)
       -> (SET-OF ?X (DIFF ?S1 ?S2) ?P)
       --
       (NOTE IDENTITY (DIFF (INTERSECT S1 P) S2)
             = (INTERSECT S1 P (COMPLEMENT S2))
             = (INTERSECT (DIFF S1 S2) P)))
Used in: ((DERIV3 : 40))

(RULE6 (SET-DEPLETION-RULE USED 2 TIMES)
       * (ACHIEVE (EMPTY ?S))
       ->
       (UNTIL (! $FILL-IN "Supergoal of emptying set")
              (ACHIEVE (REMOVE-1-FROM ?S)))
       --
       (FILL-IN SIMULATES EFFECT OF RETRIEVING SUPERGOAL FROM STORED EXPLANATION
                OF HEURISTIC))
Used in: ((DERIV3 : 38) (DERIV8 : 3))

(RULE7 (PROPAGATE-SPLIT-RULE USED 2 TIMES)
       * (REMOVE-1-FROM (SET-OF ?X ?S ?P))
       -> (UNDO (AND [IN ?X ?S] ?P)))
Used in: ((DERIV3 : 41) (DERIV8 : 4))

(RULE10 (PROPAGATE-SPLIT-RULE USED 4 TIMES)
        * (IN ?C (SET-OF ?X ?S ?P))
        -> (AND [IN ?C ?S] [! $SUBST ?C ?X ?P]))
Used in: ((DERIV2 : 91) (DERIV3 : 17) (DERIV6 : 16) (DERIV8 : 7))

(RULE11 (SIMPLIFY->NON-CONSTANT-RULE USED 8 TIMES)
        ** (?P : T)
        * (?Q : (AND & ?L))
        -> (! $REMOVE ?P ?Q))
Used in: ((DERIV2 : 33 - 34 ; 53 60 86)
          (DERIV3 : 23 29 82))

(RULE12 (SIMPLIFY->NON-CONSTANT-RULE USED 5 TIMES)
        * (=> T ?P)
        -> ?P)
Used in: ((DERIV3 : 86) (DERIV13 : 26 50 70 102))

(RULE13 (TRANSFER-RULE USED 1 TIMES)
        *
        ((?Q : QUANTIFIER PREDICATE)
         ?X ((?M : INJECTIVE MAPPED QUANTIFIER)
             ?Y ?S ?E)
         ?P)
        -> (?Q ?Y ?S (! $SUBST ?E ?X ?P)))
```

```
Used in: ((DERIV13 : 30))

(RULE14 (ELABORATE-RULE USED 1 TIMES)
        * (IN ?E (PROJECT ?F ?S))
        ->
        (EXISTS (?X : (! Q*V (! $FILL-IN "variable name")))
                ?S (= ?E (?F ?X))))
Used in: ((DERIV4 : 45))

(RULE15 (ELABORATE-RULE MAKE-ASSUMPTION-RULE USED 1 TIMES)
        * (PARTITION & ?L)
        -> (! $ASSUMING (DISJOINT & ?L) T (UNION & ?L)))
Used in: ((DERIV4 : 10))

(RULE17 (PROPAGATE-SPLIT-RULE USED 1 TIMES)
        * (SUBSET (SET & ?L) ?S)
        -> (AND & [! $DISTRIBUTE ?XX ?L (IN ?XX ?S)]))
Used in: ((DERIV3 : 14))

(RULE19 (SPECIAL-CASE-REDUCTION-RULE USED 1 TIMES)
        * (?D : (DIFF (JOIN & ?L) ?S))
        -> (! $TRY ?D (DISJOINT & ?L) (JOIN & (! $REMOVE ?SS ?L)))
        IF
        (AND [! $IN! (!! ?SS : ?S) ?L] [! $TRYABLE! ?D (DISJOINT & ?L)])
        -- (?SS HACK IS TO MAKE $REMOVE WORK AS DESIRED)
        ((DIFF (JOIN ?S & ?L) ?S) -> (JOIN & ?L)))
Used in: ((DERIV4   16))

(RULE30 (PARTIAL-MATCH-RULE USED 3 TIMES)
        * (?E : ((?R : TRANSITIVE)
                 (?F ?S1)
                 (?F (?S2 : $SET!))))
        -> (! $SUFFICE ?E (SUBSET ?S1 ?S2))
        -- (ASSUMING (SUBSET ?S1 ?S2) => (?R (?F ?S1) (?F ?S2))))
Used in: ((DERIV13 : 23 46 96))

(RULE32 (MAKE-ASSUMPTION-RULE USED 3 TIMES)
        * (?G : (SHOW ?Q))
        -> (! $MARK (ASSUMING ?Q) (! $RETURN ?B ?E))
        IF ($TRIED! ?B ?P ?E ?G)
        -- (OPTIMISTIC CASE: ASSUME ?Q IS TRUE))
Used in: ((DERIV2 : 64 67) (DERIV9 : 37))

(RULE34 (CONTROL-RULE USED 14 TIMES)
        * (?G : (SHOW T))
        -> (! $MARK (FACT ?B -> ?E) (! $RETURN ?B ?E))
        IF ($TRIED! ?B ?P ?E ?G)
        --
        (TEST SHOULD INCORPORATE OTHER CONSTRAINTS IN ORIGINAL RULE AND DEPEND
             ON STATE DEPENDENCIES AND LIMITATIONS OF RESULT DERIVATION)
        (RECORD SPECIAL CASE RULE ?B -> ?E))
Used in: ((DERIV2 : 85 100)
          (DERIV3 : 59)
          (DERIV4 : 18)
          (DERIV5 : 49)
          (DERIV6 : 19 49)
          (DERIV9 : 14)
          (DERIV10 : 12)
          (DERIV11 : 5)
          (DERIV12 : 13)
          (DERIV13 : 39 76)
          (DERIV14 : 57))

(RULE35 (PURE-TRANSPOSE-RULE PROPAGATE-SPLIT-RULE USED 5 TIMES)
        * (AFFECT (AND & ?L))
        -> (AND [AFFECT (?P : (! $SELECT ?L))] & [! $REMOVE ?P ?L])
        -- (SEE RULE260)
        (SUFFICIENT CONDITION -- NECESSARY IF ?P IS ONLY CHANGEABLE CONJUNCT))
Used in: ((DERIV3 : 42 44) (DERIV8 : 5 8) (DERIV12 : 7))

(RULE43 (PARTIAL-MATCH-RULE USED 6 TIMES)
        * ((?R : REFLEXIVE)
          ((?F : (~ QUANTIFIER)) & ?L)
          (?F & ?M))
        -> (AND & [! $DISTRIBUTEL (?XI ?YI) (?L ?M) (= ?XI ?YI)])
        IF ($EQUAL-LENGTH! ?L ?M)
        --
        (SUFFICIENT CONDITION -- NECESSARY IF (R (F X) (F Y))
                    => X = Y))
Used in: ((DERIV3 : 52 56)
          (DERIV5 : 12)
          (DERIV6 : 11)
          (DERIV9 : 33)
          (DERIV11 : 11))

(RULE47 (OPPORTUNISTIC-EVALUATION-RULE USED 2 TIMES)
        * (=> ?P T)
        -> T)
Used in: ((DERIV3 : 28 33))

(RULE52 (CONTROL-RULE USED 1 TIMES)
        * (?G : (SHOW NIL))
        -> (! $RETURN ?P NIL)
        IF ($CHECKED! ?P ?Q ?G))
Used in: ((DERIV14 : 76))

(RULE53 (CONTROL-RULE USED 12 TIMES)
        * (?G : (SHOW T))
        -> (! $RETURN ?P T)
        IF ($SUFFICED! ?P ?Q ?G))
Used in: ((DERIV2 : 52 99)
          (DERIV3 : 75)
          (DERIV6 : 58)
          (DERIV11 : 4)
          (DERIV13 : 25 48 - 49 ; 69 99 - 101))

(RULE57 (INTERSECTION-SEARCH-RULE USED 4 TIMES)
        * (DURING ?X ?Y)
        -> NIL IF ($DISJOINT! (! $PARTS<EVENT ?X) (! $GENL<EVENT ?Y)))
Used in: ((DERIV5 : 8) (DERIV6 : 5) (DERIV10 : 10) (DERIV12 : 11))

(RULE58 (QUASI-TRANSPOSE-RULE USED 2 TIMES)
        * (DURING (EACH ?X ?S ?F) ?E)
        -> (EXISTS ?X ?S (DURING ?F ?E)))
Used in: ((DERIV5 : 11) (DERIV6 : 9))

(RULE59 (PROPAGATE-SPLIT-RULE USED 2 TIMES)
        * (EXISTS ?X ?S (?P : (AND & ?L)))
        -> (AND [IN ?C ?S] & [! $SUBST ?C ?X (! $REMOVE ?Q ?L)])
        IF
        (OR [! $IN! (!! ?Q : (= ?X ?C)) ?L]
            [! $IN! (!! ?Q : (= ?C ?X)) ?L]))
Used in: ((DERIV5 : 13) (DERIV6 : 12))

(RULE61 (FACT-RULE USED 1 TIMES)
        * (ACTIONS-OF ?P)
        -> (PLAY-CARD ?P)
        IF (TRICK-IN-PROGRESS))
Used in: ((DERIV3 : 5))

(RULE62 (FACT-RULE USED 1 TIMES) * (IN QS (CARDS)) -> T)
Used in: ((DERIV3 : 18))

(RULE64 (PROPAGATE-SPLIT-RULE USED 3 TIMES)
        * (= ?C (THE ?X ?S ?P))
        -> (AND [IN ?C ?S] [! $SUBST ?C ?X ?P]))
Used in: ((DERIV2 : 77) (DERIV5 : 19) (DERIV6 : 32))

(RULE84 (EXAMPLE-RULE USED 1 TIMES)
        * (?E : (FIND-ELT ?S))
        -> (! $TRY ?E (IN ?C ?S) ?C)
        IF
        ($EXISTS! (!! IN (?C : $CONSTANT!) ?L)
                  (! $PREMISES-OF ?E)
```

```
                (!! $TRYABLE! ?E (IN ?C ?S))))
Used in: ((DERIV5 : 39))

(RULE88 (SIMPLIFY->NON-CONSTANT-RULE USED 3 TIMES)
        * (NOT (NOT ?P))
        -> ?P)
Used in: ((DERIV2 : 43) (DERIV5 : 38) (DERIV6 : 41))

(RULE91 (PARTIAL-MATCH-RULE USED 3 TIMES)
        * (?E : ((?R : REFLEXIVE) (?F & ?L) (?F & ?M)))
        ->
        (! $SUFFICE ?E
           (AND & [! $DISTRIBUTEL (?XI ?YI) (?L ?M) (= ?XI ?YI)]))
        IF ($EQUAL-LENGTH! ?L ?M))
Used in: ((DERIV2 : 48 62) (DERIV6 : 54))

(RULE100 (QUASI-TRANSPOSE-RULE USED 1 TIMES)
         * (DURING ?T (FOR-SOME ?X ?S ?E))
         -> (EXISTS ?X ?S (DURING ?T ?E)))
Used in: ((DERIV6 : 10))

(RULE103 (PURE-TRANSPOSE-RULE USED 1 TIMES)
         * (?E : ((?Q : QUANTIFIER PREDICATE)
                 ?X ?S (AND & ?L)))
         -> (AND ?P [?Q ?X ?S (AND & [! $REMOVE ?P ?L])])
         IF ($EXISTS! ?P ?L (!! $INDEPENDENT-OF ?P ?X)))
Used in: ((DERIV6 : 13))

(RULE109 (EXAMPLE-RULE USED 2 TIMES)
         * (NOT (AND & ?L))
         ->
         (=> (AND & [! $REMOVE ?E ?L])
             (NOT (! $SUBST (FIND-ELT ?S) ?X ?Q)))
         IF ($IN! (!! ?E : (FORALL ?X ?S ?Q)) ?L))
Used in: ((DERIV5 : 35) (DERIV6 : 40))

(RULE121 (PROPAGATE-SPLIT-RULE USED 2 TIMES)
         * (SET-OF ?X (COLLECTION & ?L) ?P)
         ->
         (UNION & (! $DISTRIBUTE ?Y ?L (!! IF (! $SUBST ?Y ?X ?P)
                                              (SET ?Y)
                                              NIL))))
Used in: ((DERIV14 : 20 64))

(RULE123 (RECOGNIZE-RULE USED 42 TIMES)
         * ?E -> (! $RECOGNIZE ?E (! $SELECT ?L))
         IF
         ($NON-EMPTY! (?L :
                          (! $SET-OF (!! ?F)
                             (! $FNS<E ?E)
                             (!! ! $RECOGNIZABLE! ?E ?F)))))
Used in: ((DERIV2 : 17)
          (DERIV3 : 2 - 3 ; 50 87 93 - 94 ; 96 - 97)
          (DERIV4 : 14 34 - 37 ; 40 - 42 ; 46 52 - 54)
          (DERIV5 : 52 - 53)
          (DERIV6 : 21 - 22)
          (DERIV8 : 11 14)
          (DERIV9 : 12 38 - 39 ; 41 - 43)
          (DERIV10 : 9 30)
          (DERIV11 : 18)
          (DERIV12 : 10)
          (DERIV13 : 14)
          (DERIV14 : 15 17 43 79))

(RULE124 (ELABORATE-RULE USED 119 TIMES)
         * (?E : $EXPANDABLE!)
         -> (! $EXPAND-DEF ?E))
Used in: ((DERIV2 : 1 - 2 ; 5 11 24 27 29 41 - 42 ; 44 75 - 76 ; 78 90 103)
          (DERIV3 : 1 4 6 16 20 - 21 ; 24 39 43 47 - 48 ; 51 65 - 66 ; 68 71 83
                  90 92)
          (DERIV4 : 1 - 4 ; 11 22 44)
          (DERIV5 : 2 6 10 14 16 - 18 ; 20 22 26 - 27 ; 29 - 30 ; 40 44)
          (DERIV6 : 1 - 3 ; 7 - 8 ; 15 24 - 25 ; 29 - 31 ; 34 46 52 - 53)
          (DERIV7 : 1 - 2)
          (DERIV8 : 1 6 9)
          (DERIV9 : 1 3 - 4 ; 8 23 - 24 ; 29 31 40 49)
          (DERIV10 : 1 - 3 ; 5 7 26 28)
          (DERIV11 : 7)
          (DERIV12 : 2 4 8)
          (DERIV13 : 3 5 11 - 12 ; 16 29 60 - 61 ; 84 94 - 95 ; 112)
          (DERIV14 : 10 - 11 ; 19 29 32 41 53 63 70 85))

(RULE127 (VARIABLE-RULE USED 15 TIMES)
         * (?X : $BOUND-VAR!)
         -> (! $BINDING<VAR ?X))
Used in: ((DERIV2 : 36 38 55 88)
          (DERIV3 : 60 - 61)
          (DERIV6 : 51 60)
          (DERIV9 : 18 20)
          (DERIV11 : 6 15 - 16)
          (DERIV13 : 41 78))

(RULE131 (PROPAGATE-SPLIT-RULE USED 2 TIMES)
         * (= (THE ?X ?S ?P) ?C)
         -> (AND [IN ?C ?S] [! $SUBST ?C ?X ?P]))
Used in: ((DERIV6 : 26) (DERIV13 : 62))

(RULE135 (TRANSFER-RULE USED 3 TIMES)
         * ((?Q : EXISTENTIAL QUANTIFIER)
            ?X (SET-OF ?Y ?S ?P)
            ?R)
         -> (?Q ?X ?S (AND [! $SUBST ?X ?Y ?P] ?R))
         --
         (WORKS FOR ?Q IN (SET-OF THE EXISTS)
               BUT NOT (FORALL FOR-SOME EACH DISTRIBUTE CHOOSE)))
Used in: ((DERIV2 : 45) (DERIV3 : 67) (DERIV12 : 5))

(RULE142 (EXAMPLE-RULE USED 1 TIMES)
         ** (?E : (FIND-ELT ?S))
         * (?P : (PREDICATE & ?ETC))
         ->
         (! $SUFFICE ?P
            (AND [IN (?C : (! Q*V (! $FILL-IN "instance name"))) ?S]
                 [! $SUBST ?C ?E ?P])))
Used in: ((DERIV6 : 44))

(RULE146 (FACT-RULE USED 1 TIMES)
         * (= (PLAYER-OF ?C) ?P)
         -> (= (CARD-OF ?P) ?C))
Used in: ((DERIV5 : 15))

(RULE149 (PROPAGATE-SPLIT-RULE USED 2 TIMES)
         * (IN ?C ((?A : $FILTER!) ?S))
         -> (AND [! $PRED<ADJ ?A ?C] [IN ?C ?S]))
Used in: ((DERIV6 : 33 45))

(RULE151 (REDUCE-SUFF-COND-RULE USED 3 TIMES)
         * (?G : (SHOW ?X))
         -> (! $MARK (FAILED ?Q ?G) (! $RETURN ?P ?X))
         IF ($SUFFICED! ?P ?Q ?G)
         -- (RETURN REDUCED RESULT -- USEFUL AS GOAL))
Used in: ((DERIV2 : 37 72) (DERIV6 : 61))

(RULE153 (LATERAL-REPHRASE-RULE USED 4 TIMES)
         * ((?R : COMPARATIVE) ?E1 ?E2)
         -> ((! $OPPOSITE-OF ?R) ?E2 ?E1))
Used in: ((DERIV6 : 42 62) (DERIV7 : 5) (DERIV13 : 87))

(RULE154 (APPROXIMATION-RULE USED 2 TIMES)
         * ((?R : COMPARATIVE) ?C ?X)
         -> ((! $PRED<COMP ?R) ?C)
         -- (USE UNARY PREDICATE TO APPROXIMATE COMPARISON WITH UNKNOWN))
Used in: ((DERIV6 : 43) (DERIV7 : 6))
```

```
(RULE155 (SIMPLIFY->NON-CONSTANT-RULE USED 2 TIMES)
         • ((?Q : QUANTIFIER PREDICATE) ?X ?S ?P)
         -> ?P IF ($INDEPENDENT-OF ?P ?X))
Used in: ((DERIV2 : 56) (DERIV7 : 7))

(RULE156 (CONSTRAIN-CHOICE-RULE USED 0 TIMES)
         • (ACHIEVE (?P : ((?F : PREDICATE) & ?L)))
         ->
         (CHOOSE ?X (SET-OF (?Y : (! $PRIME ?X)) ?S (! $SUBST ?Y ?X ?P))
                 ?E)
         IF ($EXISTSP! ?X ?L (!! ! $CHOSEN! ?X ?S ?E))
         -- (P MUST EXPLICITLY MENTION X)
         (IMPLICITLY USED IN DERIV6)
         (TO ACHIEVE P (X) CONSTRAIN CHOICE E (X) TO SATISFY P (X)))

(RULE157 (MAKE-ASSUMPTION-RULE USED 1 TIMES)
         • (=> ?P ?Q)
         -> ?Q -- (ASSUME ?P IN (ACHIEVE (=> ?P ?Q))))
Used in: ((DERIV5 : 50))

(RULE159 (LATERAL-REPHRASE-RULE USED 1 TIMES)
         • (NOT (EXISTS ?X ?S ?P))
         -> (EMPTY (SET-OF ?X ?S ?P)))
Used in: ((DERIV8 : 2))

(RULE161 (TRANSFER-RULE USED 1 TIMES)
         • ((?Q : QUANTIFIER) ?X ?S ?P)
         -> (?Q (?Y : (! Q*V Y)) (PROJECT ?F ?S) (! $SUBST ?Y ?E ?P))
         IF ($IN! (?E : (?F ?X)) ?P))
Used in: ((DERIV4 : 5))

(RULE162 (RECOGNIZE-RULE USED 2 TIMES)
         • (EXISTS ?X ?S (= ?E ?X))
         -> (IN ?E ?S)
         --
         (RECOGNIZING (= ?E ?X)
                          AS A HIGHER-LEVEL CONCEPT MAY BE BETTER))
Used in: ((DERIV4 : 6) (DERIV9 : 36))

(RULE163 (INTRODUCE-SPLIT-RULE FACT-RULE USED 1 TIMES)
         • (RANGE LOC)
         ->
         (PARTITION (HANDS (PLAYERS))
                    (PILES (PLAYERS))
                    (SET DECK POT HOLE))
         -- (ACTUALLY PARTITION -- ENUMERATE CASES NOT ELEMENTS))
Used in: ((DERIV4 : 9))

(RULE164 (INTRODUCE-SPLIT-RULE USED 1 TIMES)
         • (DIFF ?S1 (DIFF ?S2 ?S3))
         -> (UNION (DIFF ?S1 ?S2) (INTERSECT ?S1 ?S3)))
Used in: ((DERIV4 : 15))

(RULE165 (INTRODUCE-SPLIT-RULE USED 1 TIMES)
         • (INTERSECT ?S (SET ?C))
         -> (IF (IN ?C ?S) (SET ?C) NIL))
Used in: ((DERIV4 : 20))

(RULE169 (PIGEON-HOLE-RULE USED 1 TIMES)
         • (?P : (IN (?E : (?F & ?L)) ?S))
         -> (NOT (! $MARK INVERTED (IN ?E (DIFF (RANGE ?F) ?S))))
         IF (NOT (! $MARKED! INVERTED ?P)))
Used in: ((DERIV4 : 7))

(RULE170 (PURE-TRANSPOSE-RULE USED 1 TIMES)
         • (PROJECT ?F (DIFF ?S1 ?S2))
         -> (DIFF (PROJECT ?F ?S1) (PROJECT ?F ?S2))
         -- (ASSUMING ?F IS INJECTIVE))
Used in: ((DERIV4 : 12))

(RULE171 (QUASI-TRANSPOSE-RULE USED 1 TIMES)
         • (PROJECT ?F (SET & ?L))
         -> (SET & (! $DISTRIBUTE ?X ?L (?F ?X))))
Used in: ((DERIV4 : 13))

(RULE172 (PARTIAL-MATCH-RULE USED 3 TIMES)
         * (IN (?F ?E) (PROJECT ?F ?S))
         -> (IN ?E ?S)
         -- (SUFFICIENT CONDITION -- NECESSARY IF ?F IS INJECTIVE))
Used in: ((DERIV4 : 23) (DERIV5 : 23) (DERIV6 : 35))

(RULE173 (FACT-RULE USED 4 TIMES) * (IN ME (PLAYERS)) -> T)
Used in: ((DERIV4 : 24) (DERIV5 : 24) (DERIV6 : 27 36))

(RULE176 (SIMPLIFY->NON-CONSTANT-RULE USED 5 TIMES)
         ** (?X : NIL)
         * (?E : (JOIN & ?ETC))
         -> (! $REMOVE ?X ?E))
Used in: ((DERIV2 : 8)
          (DERIV3 : 78)
          (DERIV4 : 39 49)
          (DERIV10 : 18))

(RULE177 (SIMPLIFY->NON-CONSTANT-RULE USED 12 TIMES)
         ** (?P : ((?O : COMM-CONN) & ?L))
         * (?Q : (?O & ?M))
         -> (?O & ?L & (! $REMOVE ?P ?M))
         -- ((?O (?O & ?L) & ?S) -> (?O & ?L & ?S)))
Used in: ((DERIV3 : 45)
          (DERIV4 : 28 33)
          (DERIV5 : 31)
          (DERIV6 : 39)
          (DERIV11 : 17)
          (DERIV13 : 59 63 66 89 103)
          (DERIV14 : 38))

(RULE178 (SIMPLIFY->NON-CONSTANT-RULE USED 18 TIMES)
         * (COMM-CONN ?E)
         -> ?E)
Used in: ((DERIV2 : 9 35 54 61 71 87)
          (DERIV3 : 15 79)
          (DERIV4 : 50)
          (DERIV5 : 37)
          (DERIV6 : 14 20)
          (DERIV9 : 46)
          (DERIV11 : 14)
          (DERIV13 : 37 54)
          (DERIV14 : 21 65))

(RULE179 (OPPORTUNISTIC-EVALUATION-RULE USED 6 TIMES)
         * ((?R : REFLEXIVE) ?E ?E)
         -> T)
Used in: ((DERIV3 : 73)
          (DERIV5 : 33 48)
          (DERIV6 : 56)
          (DERIV9 : 34)
          (DERIV14 : 66))

(RULE180 (FACT-RULE USED 1 TIMES)
         * (AT CARD DECK)
         -> NIL IF (ROUND-PROGRESSING))
Used in: ((DERIV4 : 38))

(RULE182 (PROPAGATE-SPLIT-RULE USED 2 TIMES)
         * (IN ?C (SET & ?L))
         -> (OR & [! $DISTRIBUTE ?XX ?L (= ?C ?XX)]))
Used in: ((DERIV4 : 32) (DERIV13 : 36))

(RULE184 (OPPORTUNISTIC-EVALUATION-RULE USED 3 TIMES)
         ** T * (OR & ?L)
         -> T)
Used in: ((DERIV3 : 27 32) (DERIV4 : 25))

(RULE186 (SIMPLIFY->NON-CONSTANT-RULE USED 2 TIMES)
          * (IF T ?X ?Y)
          -> ?X)
Used in: ((DERIV4 : 26) (DERIV14 : 67))

(RULE187 (TRANSFER-RULE USED 1 TIMES)
          ** (?S1 : (SET & ?L1))
          * ((?U : UNION) & ?L)
          -> (?U (SET & ?L1 & ?L2) & (! $REMOVE ?S1 (! $REMOVE ?S2 ?L)))
          IF
          ($EXISTS! (!! ?S2 : (SET & ?L2))
                    (! $FOLLOWING ?S1 ?L)
                    (!! $DIFF-NODE! ?S1 ?S2))
          -- ((UNION (SET A) (SET B) &C)
              -> (UNION (SET A B) &C)))
Used in: ((DERIV4 : 29))

(RULE188 (SIMPLIFY->NON-CONSTANT-RULE USED 2 TIMES)
          * ((?O : COMM-CONN) & ?L)
          -> (?O & (! $REMOVE ?S2 ?L))
          IF
          ($EXISTS! ?S1 ?L
                    (!! $EXISTS! ?S2 (! $FOLLOWING ?S1 ?L)
                          (!! AND (! $EQ! ?S1 ?S2) (! $DIFF-NODE! ?S1 ?S2))))
          -- (ELIMINATE DUPLICATES))
Used in: ((DERIV2 : 70) (DERIV13 : 90))

(RULE189 (DETECT-APPLICABILITY DISJOINT-SUBSETS-RULE USED 1 TIMES)
          ** (?S : $SET! $UNKNOWN!)
          * (?P : ((?Q : PREDICATE (~ DISJOINT))
                 & ?M))
          -> (! $REFORMULATE ?P (DISJOINT ?S (! Q*V S)))
          IF ($TRANSFORMABLE! ?P (DISJOINT ?S ?H)))
Used in: ((DERIV9 : 2))

(RULE190 (CONTROL-RULE USED 1 TIMES)
          * (?G : (SHOW T))
          -> (! $RETURN ?P ?Q)
          IF
          (AND [! $TRANSFORMED! ?P ?Q ?G]
               [! NOT (! $INP! (!! ?X : $UNBOUND-VAR!) ?Q)]])
          -- (PROCESS OF MATCHING ?P AND ?Q SHOULD BIND VARIABLES IN ?Q))
Used in: ((DERIV9 : 17))

(RULE192 (PARTIAL-MATCH-RULE USED 3 TIMES)
          * (= ((?Q : QUANTIFIER PREDICATE) ?X ?S ?P) (?Q ?Y ?S ?R))
          -> (= (! $SUBST ?Y ?X ?P) ?R)
          -- (SUFFICIENT (NOT NECESSARY) CONDITION -- QUANTIFICATION?))
Used in: ((DERIV2 : 83) (DERIV9 : 6) (DERIV13 : 74))

(RULE193 (REDUCE->SUFF-COND-RULE USED 1 TIMES)
          * (= ?P (IN (?X : $VAR!) ?S))
          -> (= ?S (SET-OF ?X (DOMAIN ?X ?P) ?P)))
Used in: ((DERIV9 : 7))

(RULE194 (VARIABLE-RULE USED 7 TIMES)
          * (= (?X : $UNBOUND-VAR!) ?E)
          -> (! $BIND-VAR ?X ?E))
Used in: ((DERIV3 : 53 56)
          (DERIV9 : 13 16)
          (DERIV11 : 3 12)
          (DERIV13 : 38))

(RULE195 (FUNCTIONAL-DOMAIN-RULE USED 1 TIMES)
          * (DOMAIN ?X (?F & ?L))
          -> (DOMAIN ?X ?E)
          IF ($EXISTS! (!! ?E : (?G & ?M)) ?L (!! $IN! ?X ?M)))
Used in: ((DERIV9 : 9))

(RULE196 (FUNCTIONAL-DOMAIN-RULE USED 1 TIMES)
          * (DOMAIN ?X (?F ?X))
          -> (DOMAIN ?F))
Used in: ((DERIV9 : 10))
```

```
(RULE197 (FUNCTIONAL-DOMAIN-RULE FACT-RULE USED 1 TIMES)
         * (DOMAIN SUIT-OF)
         -> (CARDS))
Used in: ((DERIV9 : 11))

(RULE198 (DISJOINT-SUBSETS-RULE USED 1 TIMES)
         * (DISJOINT (?H : $UNKNOWN!) ?S)
         -> (PR-DISJOINT ?H ?S (COMMON-SUPERSET ?H ?S))
         -- (ASSUME ?H AND OR ?S CHOSEN RANDOMLY FROM ?U)
         (SIZE OF COMMON SUPERSET ?U SHOULD BE KNOWN))
Used in: ((DERIV9 : 21))

(RULE199 (INTERSECTION-SEARCH-RULE USED 1 TIMES)
         * (COMMON-SUPERSET (SET-OF ?X ?S ?P) (SET-OF ?Y ?S ?Q))
         -> ?S)
Used in: ((DERIV9 : 25))

(RULE200 (REFINE DISJOINT-SUBSETS-RULE USED 1 TIMES)
         * (?E : (PR-DISJOINT (?H : $UNKNOWN!) ?S ?U))
         ->
         (! $TRY ?E (- ?H (FILTER ?H (?P : (! $SELECT ?L))))
            (PR-DISJOINT ?H (FILTER ?S ?P) (FILTER ?U ?P)))
         IF
         ($NON-EMPTY! (?L :
                          (! $SET-OF (!! ?P : $UNARY!)
                             (! $PREDICATES)
                             (!! ! $TRYABLE! ?E (- ?H (FILTER ?H ?P))))))
         -- (IMPROVES ACCURACY BY NARROWING DOWN SETS ?S AND ?U)
         (SHOULD ALSO REQUIRE (SUBSET ?U (DOMAIN ?P))
                AND (KNOWN (# (FILTER ?U ?P)))))
Used in: ((DERIV9 : 27))

(RULE202 (FUNCTIONAL-DEPENDENCE-RULE USED 1 TIMES)
         * (EVAL (?E : ((?F : FORMULA) & ?L)))
         ->
         (FUNCTION-OF (?V : (! $FILL-IN "independent variable"))
                      (DEPENDENCE ?E ?V)))
Used in: ((DERIV9 : 44))

(RULE203 (DEPENDENCE-RULE USED 1 TIMES)
         * (DEPENDENCE (?E : (?F & ?L)) (?V : $ATOM!))
         ->
         (D+ &
             (! $DISTRIBUTE ?EI (! $SET-OF ?X ?L (!! $INP! ?V ?X))
                (!! D* (DEPENDENCE ?EI ?V)
                    (DEPENDENCE (?F & (! $ARGL<E ?E))
                                (! $CORRESPONDING (! $ARGL<E ?E) ?L ?EI))))))
Used in: ((DERIV9 : 46))

(RULE204 (DEPENDENCE-RULE USED 2 TIMES)
         * (DEPENDENCE ?E (?V : $ATOM!))
         -> NIL IF (NOT (! $INP! ?V ?E)))
Used in: ((DERIV9 : 51 56))

(RULE205 (DEPENDENCE-RULE USED 2 TIMES)
         * (DEPENDENCE ((?F : INCR1-DECR2) ?E1 ?E2) ?V)
         -> (D+ (DEPENDENCE ?E1 ?V) (D- (DEPENDENCE ?E2 ?V))))
Used in: ((DERIV9 : 50 55))

(RULE207 (DEPENDENCE-RULE USED 1 TIMES)
         * (DEPENDENCE (INCR1 ?E & ?L) (?V : $ATOM!))
         -> (DEPENDENCE ?E ?V)
         IF (NOT (! $IN! ?V ?L)))
Used in: ((DERIV9 : 54))

(RULE208 (DEPENDENCE-RULE USED 1 TIMES)
         * (DEPENDENCE ?E ?E)
         -> INCREASING)
Used in: ((DERIV9 : 57))

(RULE210 (FIX-UP FUNCTIONAL-DEPENDENCE-RULE USED 1 TIMES)
          * (FUNCTION-OF ?V ?E)
          -> (FUNCTION-OF ?W (! $SUBST INCREASING ?D ?E))
          IF
          ($EXISTSP! (!! ?D : (DEPENDENCE (?W : (?F & ?L)) ?V))
                     ?E (!! $IN! ?V ?L))
          -- (IDEALLY ?W IS AN OBSERVABLE PROPERTY OF ?V))
Used in: ((DERIV9 : 47))

(RULE211 (FUNCTIONAL-RECURRENCE-RULE USED 0 TIMES)
          *
          (?E :
              (= (?E1 : (?G & ?ETC))
                 (?E2 : ((?H : $UNBOUND-VAR!)
                         (?W : (?F & ?L))))))
          ->
          (! $TRY ?E (= ?V ?W)
             (= ?H (LAMBDA ((?XX : (! Q*V ?F))) (! $SUBST ?XX ?V ?E1))))
          IF
          ($EXISTSP! (!! ?V : (?F & ?M))
                     ?E1 (!! $TRYABLE! ?E (= ?V ?W)))
          -- (?W MATCHES SUB-EXPRESSION ?V OF ?E1)
          (= (?G -- ?V --) (?H ?W)))

(RULE212 (FUNCTIONAL-RECURRENCE-RULE USED 1 TIMES)
          *
          (?E :
              (= (?E1 : (?F & ?L))
                 (?E2 : ((?H : $UNBOUND-VAR!)
                         (?W : (?G & ?ETC))))))
          ->
          (! $TRY ?E (= ?E1 ?V)
             (= ?H (INVERSE (LAMBDA ((?XX : (! Q*V ?F))) (! $SUBST ?XX ?V ?W)))))
          IF
          ($EXISTSP! (!! ?V : (?F & ?M))
                     ?W (!! $TRYABLE! ?E (= ?E1 ?V)))
          -- (?E1 MATCHES SUB-EXPRESSION ?V OF ?W)
          (= ?E1 (?H (?G -- ?V --))))
Used in: ((DERIV9 : 5))

(RULE213 (HEURISTIC-EQUIVALENCE-PRESERVING-RULE USED 1 TIMES)
          * ((LAMBDA (& ?ARGS) ?BODY) & ?L)
          -> (! $SUBSTL ?L ?ARGS ?BODY)
          IF ($EQUAL-LENGTH! ?ARGS ?L))
Used in: ((DERIV14 : 86))

(RULE215 (HEURISTIC-EQUIVALENCE-PRESERVING-RULE USED 1 TIMES)
          * (LAMBDA (?X) (?F ?X))
          -> ?F)
Used in: ((DERIV9 : 15))

(RULE216 (SIMPLIFY->NON-CONSTANT-RULE USED 1 TIMES)
          ** (?W : ((INVERSE ?F) ?X))
          * (?F ?W)
          -> ?X)
Used in: ((DERIV9 : 19))

(RULE218 (LATERAL-REPHRASE-RULE USED 1 TIMES)
          * (= ?S (FILTER ?S ?P))
          -> (=> (IN (?X : (! Q*V X)) ?S) (?P ?X)))
Used in: ((DERIV9 : 28))

(RULE219 (TRANSFER-RULE USED 1 TIMES)
          * (FORALL ?X1 (SET-OF ?X2 ?S ?P) ?Q)
          -> (FORALL ?X1 ?S (=> (! $SUBST ?X1 ?X2 ?P) ?Q)))
Used in: ((DERIV2 : 79))

(RULE220 (PURE-TRANSPOSE-RULE USED 1 TIMES)
          * (=> ?P ((?F : QUANTIFIER) ?X ?S ?Q))
          -> (?F ?X ?S (=> ?P ?Q))
          -- (SPECIAL CASE OF RULE329))
Used in: ((DERIV2 : 46))
```

```
(RULE221 (MAKE-ASSUMPTION-RULE USED 1 TIMES)
         * (?G : (SHOW ?X))
         -> (! $ASSUMING ?X T (! $RETURN ?P T))
         IF ($SUFFICED! ?P ?Q ?G))
Used in: ((DERIV13 : 24))

(RULE224 (REDUCE->SUFF-COND-RULE USED 1 TIMES)
         * (=> (AND & ?L) (?P : (?F & ?M)))
         -> (=> ?Q ?P)
         IF ($EXISTS! ?Q ?L (!! $INP! ?F ?Q)))
Used in: ((DERIV11 : 8))

(RULE225 (REDUCE->SUFF-COND-RULE USED 1 TIMES)
         * (=> (?C : (OR & ?L)) (?P : (?F & ?M)))
         -> (AND [NOT (! $REMOVE ?Q ?C)] [=> ?Q ?P])
         IF ($EXISTS! ?Q ?L (!! $INP! ?F ?Q)))
Used in: ((DERIV11 : 10))

(RULE226 (REDUCE->SUFF-COND-RULE USED 1 TIMES)
         * (=> (=> ?C ?Q) (?P : (?F & ?M)))
         -> (AND ?C [=> ?Q ?P])
         IF ($INP! ?F ?Q))
Used in: ((DERIV11 : 9))

(RULE227 (SUFFICIENT-CONDITION-RULE USED 1 TIMES)
         * (?E : (EVAL (?Q : ((?F : PREDICATE) & ?ETC))))
         ->
         (! $TRY ?E (?R : (! $INSTANTIATE (! $SELECT ?S)))
            (EVAL (=> ?R ?Q)))
         IF
         ($NON-EMPTY! (?S :
                           (! $SET-OF ?P (! $PREDICATES)
                              (!! $INP! ?F (! $BODY<FN ?P))))))
Used in: ((DERIV11 : 1))

(RULE228 (FACT-RULE USED 1 TIMES)
         * (?E : (LEGAL ?P ?C))
         -> (! $SUFFICE ?E (= ?C (CARD-OF ?P)))
         IF ($TRYABLE! ?E (= ?C (CARD-OF ?P)))
         -- ((LEGAL P (CARD-OF P)) IS A LEMMA))
Used in: ((DERIV11 : 2))

(RULE234 (HISTORY-RULE USED 1 TIMES)
         * (?P : (PREDICATE & ?ETC))
         ->
         (! $TRY ?P
            (NOT (DURING (! $INSTANTIATE (?A : (! $SELECT (: $ACTIONS))))
                         (UNDO ?P)))
            (WAS-DURING (CURRENT ?A) ?P)))
Used in: ((DERIV12 : 1))

(RULE236 (SYSTEMATIC-EVALUATION-RULE USED 18 TIMES)
         * ((?F : COMPUTABLE) & ?L)
         -> (! $APPLY ?F ?L)
         IF ($CONSTANTL! ?L))
Used in: ((DERIV2 : 51)
          (DERIV3 : 31 34 57 - 58 ; 74 76 85)
          (DERIV6 : 57)
          (DERIV9 : 52 58 - 59)
          (DERIV10 : 11)
          (DERIV12 : 12)
          (DERIV13 : 34)
          (DERIV14 : 34 75 77))

(RULE238 (REDUCE->NON-EQUIVALENT-RULE USED 1 TIMES)
         * (?C : (CHOOSE ?X ?S ?E))
         -> (! $MARK (CHOOSE ?C ?S ?E) ?E)
         -- (SUBSUMES RULE132 AND RULE158))
Used in: ((DERIV13 : 13))

(RULE253 (SIMPLIFY->NON-CONSTANT-RULE USED 1 TIMES)
         * (LB:UB ?L ?L)
         -> (SET ?L))
Used in: ((DERIV13 : 35))

(RULE254 (ACTION-RULE USED 1 TIMES)
         • (?E : (CHANGE ?P))
         ->
         (! $TRY ?E
            (=> (?ACTION : (! $INSTANTIATE (! $SELECT (! $ACTIONS)))) ?E)
            ?ACTION)
         IF
         ($EXISTS! ?ACT (! $ACTIONS)
                   (! $TRYABLE! ?E (=> (?ACT & ?ETC) ?E))))
Used in: ((DERIV3 : 46))

(RULE255 (SIMPLIFY->NON-CONSTANT-RULE USED 1 TIMES)
         • ((?Q : QUANTIFIER PREDICATE)
            ?X (COLLECTION ?C)
            ?P)
         -> (! $SUBST ?C ?X ?P))
Used in: ((DERIV2 : 104))

(RULE256 (INTRODUCE-SPLIT-RULE DESIGN-CHOICE-RULE USED 1 TIMES)
         • (?P : (PREDICATE & ?ETC))
         ->
         (! $ASSUMING
            (SELECT (?X : (! Q*V (?E : (! $SELECT ?L)))) (RANGE ?E))
            T (AND [= ?E ?X] [! $SUBST ?X ?E ?P]))
         IF ($NON-EMPTY! (?L : (! $SET-OFP (!! ?F & ?ARGS) ?P T)))
         --
         (CHOOSE VALUE ?X FOR ?E BEFORE CONSTRUCTING THE OBJECT ON WHICH ?E
                 DEPENDS))
Used in: ((DERIV13 : 58))

(RULE258 (INTRODUCE-SPLIT-RULE TEMPORAL-GOAL-RULE USED 0 TIMES)
         • ?P -> (AND [=> (NOT ?P) (CAUSE ?P)] [=> ?P (NOT (UNDO ?P))])
         -- (COULD HAVE BEEN USED IN DERIV14)
         (TO ACHIEVE GOAL P AT END OF INTERVAL))

(RULE266 (TEMPORAL-GOAL-RULE USED 1 TIMES)
         • (?P : (ACHIEVE (PREDICATE & ?ETC)))
         ->
         (FORALL (?T : (! Q*V T))
                 (LB:UB 0 (- (# ?S) 1))
                 ((?Q :
                      (! $SUBST (LAMBDA ((?J : (! Q*V J))) (PREFIX ?S ?J))
                         ?S ?P))
                  ?T))
         IF ($INP! (!! ?S : SEQUENCE) ?P)
         --
         (CONVERT STATIC SEQUENCE CONSTRAINT P TO TIME-VARYING GOAL PREDICATE Q)
 )
Used in: ((DERIV14 : 1))

(RULE267 (RECOGNIZE-RULE USED 3 TIMES)
         • (LAMBDA ?ARGS ?BODY)
         -> ?F IF
         ($EXISTS! ?F (! $FNS<E ?BODY) (!! ! $RECOGNIZABLE! ?BODY ?F)))
Used in: ((DERIV14 : 3 - 4 ; 14))

(RULE268 (CONTROL-RULE USED 20 TIMES)
         • ?E -> (! $CONSIDER ?E)
         -- (FOCUS MECHANISM FOR CONVENIENCE))
Used in: ((DERIV3 : 13 64)
          (DERIV4 : 8 19 43)
          (DERIV5 : 5 36)
          (DERIV6 : 23)
          (DERIV9 : 22)
          (DERIV10 : 13 25)
          (DERIV13 : 82 110)
          (DERIV14 : 2 9 28 39 - 40 ; 52 83))

(RULE269 (CONTROL-RULE USED 20 TIMES)
         * (?E : (CONSIDER ?N))
         -> (! $RETURN ?O ?N)
         IF ($CONSIDERED! ?E ?O)
         -- (REMEMBERS ?O -> ?N FOR LATER USE)
         (DEFOCUS MECHANISM))
Used in: ((DERIV3 : 35 77)
          (DERIV4 : 27 30 55)
          (DERIV5 : 28 51)
          (DERIV6 : 38)
          (DERIV9 : 26)
          (DERIV10 : 22 31)
          (DERIV13 : 88 115)
          (DERIV14 : 35 37 46 51 80 - 81 ; 87))

(RULE271 (VARIABLE-RULE USED 7 TIMES)
         * (= ?E (?X : $UNBOUND-VAR!))
         -> (! $BIND-VAR ?X ?E)
         -- (MIRROR OF RULE194))
Used in: ((DERIV2 : 31 - 32 ; 49 - 50 ; 84)
          (DERIV3 : 54)
          (DERIV13 : 75))

(RULE273 (DEFUNCTIONALIZE-RULE USED 1 TIMES)
         * (?F ?X)
         -> (! $SUBSTL (! $DISTRIBUTE ?G ?S (?G ?X)) ?S ?F)
         IF ($NON-EMPTY! (?S : (! $SET-OFP (!! ?H : $UNAPPLIED!) ?F T)))
         -- (APPLY FUNCTIONS G CONTAINED IN FUNCTIONAL F TO ARGUMENT X))
Used in: ((DERIV14 : 82))

(RULE274 (INTRODUCE-SPLIT-RULE USED 1 TIMES)
         * (PREFIX ?S (+ ?I 1))
         -> (APPEND (PREFIX ?S ?I) (LIST (NTH ?S (+ ?I 1)))))
Used in: ((DERIV14 : 12))

(RULE276 (INTRODUCE-SPLIT-RULE TEMPORAL-GOAL-RULE USED 1 TIMES)
         * (?P : (PREDICATE & ?ETC))
         ->
         (OR [AND [NOT (CHANGE (?E : (! $CADR (?N : (! $SELECT ?S)))))]
                  [! $SUBST ?E ?N ?P]]
             [AND [CHANGE ?E] ?P])
         IF ($NON-EMPTY! (?S : (! $SET-OFP (!! ?N : (NEXT ?F)) ?P T)))
         -- (SPLIT ?P INTO 2 CASES DEPENDING ON WHETHER ?E CHANGES))
Used in: ((DERIV14 : 8))

(RULE277 (REDUCE->SUFF-COND-RULE USED 1 TIMES)
         * (CHANGE (?E : ((?F : EXTREME) ?S)))
         -> ((! $COMPARATIVE-OF ?F) (?F (CHANGE ?S)) ?E)
         -- (EG (MAX S) CHANGES IFF (MAX (CHANGE S)) > (MAX S)))
Used in: ((DERIV14 : 42))

(RULE278 (PURE-TRANSPOSE-RULE USED 0 TIMES)
         * (CHANGE ((?F : $PROJECTION!) ?S))
         -> (?F (CHANGE ?S))
         -- (ASSUMING (F (CHANGE S)) AND (F (OLD S)) ARE DISJOINT))

(RULE279 (FACT-RULE USED 2 TIMES)
         * (CHANGE CANTUS-1)
         -> (LIST (NEXT NOTE))
         -- (SHORTCUT -- COULD BE DERIVED FROM DEFINITION OF CANTUS-1))
Used in: ((DERIV14 : 44 58))

(RULE280 (PURE-TRANSPOSE-RULE USED 0 TIMES)
         * ((?F : $PROJECTION!) (COLLECTION & ?L))
         -> (COLLECTION & (! $DISTRIBUTE ?X ?L ((! $SINGULAR-OF ?F)
                                                ?X))))

(RULE281 (SIMPLIFY->NON-CONSTANT-RULE USED 2 TIMES)
         * (ONE-OF (COLLECTION ?C))
         -> ?C)
Used in: ((DERIV14 : 45 59))

(RULE282 (TEMPORAL-GOAL-RULE USED 1 TIMES)
         * (ACHIEVE ?P)
         -> (NEXT ?P))
Used in: ((DERIV14 : 5))

(RULE283 (PURE-TRANSPOSE-RULE USED 2 TIMES)
         * (NEXT (?E : (?G & ?L)))
         -> (! $SUBSTL (! $DISTRIBUTE ?F ?S (NEXT ?F)) ?S ?E)
         IF ($NON-EMPTY! (?S : (! $SET-OF (!! ?H : $FUNCTIONAL!) ?L T))))
Used in: ((DERIV14 : 6 - 7))

(RULE284 (QUASI-TRANSPOSE-RULE PROPAGATE-SPLIT-RULE USED 9 TIMES)
         ** (?S : (JOIN & ?L))
         * (?E : ((?F : HOMOMORPHISM) & ?M))
         ->
         ((! $ADD-OF ?F)
          & (! $DISTRIBUTE ?X ?L (!! ! $SUBST ?X ?S ?E)))
         --
         ((F (+ X Y))
          -> (+ (F X) (F Y))
          WHERE + OPERATIONS MAY BE DIFFERENT FOR (DOMAIN F)
          AND (RANGE F)))
Used in: ((DERIV2 : 6 102)
          (DERIV4 : 21 31)
          (DERIV5 : 7)
          (DERIV6 : 4)
          (DERIV13 : 106)
          (DERIV14 : 18 62))

(RULE285 (FACT-RULE USED 0 TIMES)
         * (NEXT CANTUS-1)
         -> (APPEND CANTUS-1 (LIST (NEXT NOTE)))
         -- (THIS CAN BE DERIVED FROM DEFINITION OF NEXT))

(RULE287 (PROPAGATE-SPLIT-RULE USED 2 TIMES)
         * ?E -> (IF ?B (! $SUBST ?X ?C ?E) (! $SUBST ?Y ?C ?E))
         IF ($INP! (!! ?C : (IF ?B ?X ?Y)) ?E))
Used in: ((DERIV14 : 22 25))

(RULE288 (OPPORTUNISTIC-EVALUATION-RULE USED 2 TIMES)
         * (# (COLLECTION & ?L))
         -> (! LENGTH ?L))
Used in: ((DERIV14 : 24 68))

(RULE289 (OPPORTUNISTIC-EVALUATION-RULE USED 1 TIMES)
         * (# NIL)
         -> 0)
Used in: ((DERIV14 : 23))

(RULE290 (HEURISTIC-EQUIVALENCE-PRESERVING-RULE USED 2 TIMES)
         *
         ((?R : NUMERICAL PREDICATE)
          (+ ?E (?C1 : $CONSTANT!))
          (?C2 : $CONSTANT!))
         -> (?R ?E ($ - ?C2 ?C1)))
Used in: ((DERIV14 : 27 69))

(RULE291 (RECOGNIZE-RULE USED 3 TIMES)
         * (= (# (SET-OF ?X ?S ?P)) 0)
         -> (NOT (EXISTS ?X ?S ?P)))
Used in: ((DERIV13 : 113) (DERIV14 : 30 71))

(RULE292 (RECOGNIZE-RULE USED 4 TIMES)
         * (EXISTS ?X ?S (= ?X ?E))
         -> (IN ?E ?S)
         -- (MIRROR OF RULE182))
Used in: ((DERIV13 : 86 114) (DERIV14 : 31 72))

(RULE293 (OPPORTUNISTIC-EVALUATION-RULE USED 1 TIMES)
         * (IN (ONE-OF ?S) ?S)
         -> T)
Used in: ((DERIV14 : 33))
```

```
(RULE294 (RECOGNIZE-RULE USED 1 TIMES)
         * (IF ?C NIL ?Y)
         -> (AND ?Y [NOT ?C]))
Used in: ((DERIV14 : 36))

(RULE296 (PURE-TRANSPOSE-RULE USED 1 TIMES)
         ** (?N1 : (NOT ?P))
         * (AND & ?L)
         -> (AND & [! $REMOVE ?N1 (! $REMOVE ?N2 ?L)] [NOT (OR ?P ?Q)])
         IF ($IN! (!! ?N2 : (NOT (?Q : (~ ?P)))) ?L))
Used in: ((DERIV14 : 47))

(RULE297 (PROBLEM-MERGING-RULE FUNCTIONALIZE-RULE USED 1 TIMES)
         * ((?B : BOOLEAN-OP) (?P ?E1 ?E2) (?Q ?E1 ?E2))
         -> ((?B ?P ?Q) ?E1 ?E2))
Used in: ((DERIV14 : 48))

(RULE298 (FUNCTIONALIZE-RULE USED 1 TIMES)
         * (NOT (?P & ?L))
         -> ((NOT ?P) & ?L))
Used in: ((DERIV14 : 49))

(RULE299 (FACT-RULE USED 1 TIMES) * (NOT (OR HIGHER =)) -> LOWER)
Used in: ((DERIV14 : 50))

(RULE300 (SPECIAL-CASE-REDUCTION-RULE TEMPORAL-GOAL-RULE USED 1 TIMES)
         * (?E : (NEXT (?O : ((?F : EXTREME) ?S))))
         ->
         (! $TRY ?E ((! $COMPARATIVE-OF ?F)
                     (?N : (?F (CHANGE ?S)))
                     ?O)
            ?N)
         -- (ASSUMES ?S IS EXPANDING))
Used in: ((DERIV14 : 55))

(RULE301 (ALL-PURPOSE-RULE USED 14 TIMES)
         * ?N ->
         (! $MARK
            (?N1 <- (?N2 : (! $FILL-IN "Equivalent node derived earlier")))
            ?N2)
         IF
         ($EQ! ?N (?N1 : (! $FILL-IN "Earlier node with same expression")))
         -- (?N2 WAS DERIVED FROM ?N1 OR VICE VERSA))
Used in: ((DERIV2 : 18 65 68 92)
          (DERIV5 : 1 42 46 - 47)
          (DERIV7 : 4)
          (DERIV10 : 14)
          (DERIV13 : 107)
          (DERIV14 : 54 60 - 61))

(RULE302 (CONTROL-RULE USED 2 TIMES)
         * ?P -> T IF ($INP! ?P (! $STATES))
         --
         (NOTE THAT $INP! TEST AS STATED WITHOUT RESTRICTIONS WILL ALWAYS
               SUCCEED)
         (VALID ONLY IF ?P IS PREMISE))
Used in: ((DERIV14 : 66 74))

(RULE303 (SPECIAL-CASE-REDUCTION-RULE USED 0 TIMES)
         * (?E : (SET-OF ?X ?S ?P))
         -> (! $TRY ?E (NOT (EXISTS ?X ?S ?P)) NIL))

(RULE304 (CHECK-NEC-COND-RULE USED 1 TIMES)
         * (?E : (IN ?X ?S))
         ->
         (! $CHECK ?E
            (NOT ((?P : (! $SELECT ?L))
                  ?X ((! $SUPERLATIVE-OF ?P) ?S))))
         IF
         ($NON-EMPTY! (?L :
                          (! $SET-OF (!! ?P : ANTI-SYMMETRIC)
                             (! $COMPARATIVES)
```

```
                              (!! ! $TRYABLE! ?E (?P & ?ETC))))))
Used in: ((DERIV14 : 73))

(RULE305 (FUNCTIONALIZE-RULE USED 1 TIMES)
         * (?F (?G ?T))
         -> ((?F ?G) ?T))
Used in: ((DERIV14 : 84))

(RULE306 (DETECT-APPLICABILITY HSM-INSTANTIATE USED 2 TIMES)
         ** ?X * (?P : ((?F : PREDICATE) & ?ETC))
         ->
         ((! Q*V HSM)
          WITH (PROBLEM : ?P)
          (OBJECT : ?X)
          (CHOICE-SEQ : (CHOICE-SEQ-OF ?X)))
         IF
         ($EXISTS! (!! ?S : $SEQUENCE!)
                   (! $PARTS<EVENT ?X)
                   (!! ! $IN! (!! ?C : $CHOICE!) (! $PARTS<EVENT ?S))))
Used in: ((DERIV2 : 3) (DERIV13 : 1))

(RULE307 (CHOICE-SEQ-RULE USED 2 TIMES)
         * (CHOICE-SEQ-OF (EACH ?X ?S ?E))
         -> (APPLY APPEND (EACH ?X ?S (CHOICE-SEQ-OF ?E))))
Used in: ((DERIV2 : 10) (DERIV13 : 4))

(RULE308 (INTERSECTION-SEARCH-RULE CHOICE-SEQ-RULE USED 1 TIMES)
         * (CHOICE-SEQ-OF ?S)
         -> NIL IF (NOT (! $IN! (!! ?E : $CHOICE!) (! $PARTS<EVENT ?S))))
Used in: ((DERIV2 : 7))

(RULE309 (CHOICE-SEQ-RULE USED 2 TIMES)
         * (CHOICE-SEQ-OF (?C : (CHOOSE & ?ETC)))
         -> (LIST ?C))
Used in: ((DERIV2 . 12) (DERIV13 : 6))

(RULE310 (SIMPLIFY->NON-CONSTANT-RULE USED 2 TIMES)
         * (APPLY APPEND (EACH ?X ?S (LIST ?E)))
         -> (EACH ?X ?S ?E))
Used in: ((DERIV2 : 13) (DERIV13 : 7))

(RULE311 (HSM-INSTANTIATE USED 2 TIMES)
         *
         (?HSM WITH (CHOICE-SEQ : (EACH ?X ?INDICES (CHOOSE (?F ?X)
                                                             ?S ?A)))
               (CHOICES : NONE))
         ->
         (?HSM WITH (SEQUENCE : ?F)
               (CHOICES : (PROJECT ?F ?INDICES))
               (CHOICE-SETS : (LAMBDA (?X) ?S))
               (INDICES : ?INDICES)
               (INDEX : ?X)
               (VARIABLES : NIL)
               (BINDINGS : NIL)
               (INITIAL-PATH : NIL)
               (COMPLETION-TEST : (LAMBDA (PATH) (= (# PATH) (# ?INDICES))))))
Used in: ((DERIV2 : 15) (DERIV13 : 9))

(RULE312 (CONTROL-RULE USED 5 TIMES)
         * ?E ->
         (! $CONSIDER-PROP ?E (?P : (! $SELECT ?L)) (! $PROP<VAR ?E ?P))
         IF ($NON-EMPTY! (?L : (! $PROPS<VAR ?E))))
Used in: ((DERIV2 : 4 23) (DERIV13 : 2 57 81))

(RULE313 (CONTROL-RULE USED 14 TIMES)
         * (?E : (CONSIDER-PROP ?V))
         -> (! $RETURN-PROP ?N ?P ?V)
         IF ($CONSIDERED-PROP! ?E ?N ?P))
Used in: ((DERIV2 : 14 20 25 58 73 105)
          (DERIV13 : 8 17 27 51 64 71 91 104))

(RULE314 (HSM-INSTANTIATE USED 2 TIMES)
         •
         (?HSM WITH (PROBLEM : ?P)
               (CHOICES : ?S)
               (REFORMULATED-PROBLEM : NONE))
         ->
         (! $CONSIDER-PROP ?HSM REFORMULATED-PROBLEM (REFORMULATE ?P ?S)))
Used in: ((DERIV2 : 16) (DERIV13 : 10))

(RULE315 (FUNCTIONAL-RECURRENCE-RULE USED 2 TIMES)
         • (REFORMULATE ?E ?X)
         -> ((LAMBDA ((?XX : (! Q•V ?X))) (! $SUBST ?XX ?X ?E))
             ?X)
         IF ($INP! ?X ?E))
Used in: ((DERIV2 : 19) (DERIV13 : 15))

(RULE316 (HSM-INSTANTIATE USED 1 TIMES)
         •
         (?HSM WITH (REFORMULATED-PROBLEM : (?P : ((LAMBDA (?S) ?E)
                                                   ?CHOICES)))
               (SEQUENCE : ?C))
         -> (?HSM WITH (REFORMULATED-PROBLEM : (! $SUBST ?S ?C ?P)))
         IF ($INP! ?C ?E))
Used in: ((DERIV2 : 21))

(RULE317 (HSM-INSTANTIATE USED 2 TIMES)
         •
         (?HSM WITH (REFORMULATED-PROBLEM : ((?P : (LAMBDA (?S) ?E))
                                             ?CHOICES))
               (SOLUTION-TEST : NONE))
         ->
         (?HSM WITH (PATH : ?S)
               (STEP-ORDER : NIL)
               (STEP-TEST : T)
               (PATH-ORDER : NIL)
               (PATH-TEST : T)
               (SOLUTION-TEST : ?P)))
Used in: ((DERIV2 : 22) (DERIV13 : 18))

(RULE318 (HSM-FIX-UP USED 1 TIMES)
         • (?HSM WITH (CHOICE-SETS : (LAMBDA (?I) (SET-OF ?X ?S ?P))))
         ->
         (! $CONSIDER-PROP ?HSM CHOICE-SETS
            (LAMBDA (?I) (SET-OF ?X ?S (POSSIBLE ?P))))
         -- (USE IF ?P IS NOT EVALUABLE))
Used in: ((DERIV2 : 26))

(RULE319 (NECESSARY-CONDITION-RULE USED 2 TIMES)
         • (?P : ((?F : PREDICATE) & ?ETC))
         ->
         (! $CONSIDER-NOTE ?P
            (=> (?Q : (! $INSTANTIATE (! $SELECT ?L))) IF (=> ?P ?Q)))
         IF
         ($NON-EMPTY! (?L :
                           (! $SET-OF ?R (! $PREDICATES)
                              (!! ! $CAN-EXPAND-TO ?R ?F))))
         -- (LOOK FOR PREDICATE ?R MENTIONING ?F (DIRECTLY OR INDIRECTLY)))
Used in: ((DERIV2 : 28 40))

(RULE320 (CONTROL-RULE USED 2 TIMES)
         • (?E : (CONSIDER-NOTE ?M))
         -> (! $RETURN-NOTE ?N ?O ?M)
         IF ($CONSIDERED-NOTE! ?E ?N ?O))
Used in: ((DERIV2 : 39 57))

(RULE321 (PURE-TRANSPOSE-RULE PROPAGATE-SPLIT-RULE USED 1 TIMES)
         • (=> (?P : (?F & ?ETC)) (AND & ?L))
         -> (AND [=> ?P ?Q] & [! $REMOVE ?Q ?L])
         IF ($EXISTS! ?Q ?L (!! $INP! ?F ?Q)))
Used in: ((DERIV2 : 47))

(RULE322 (PARTIAL-MATCH-RULE USED 1 TIMES)
         • (?E : (=> (?F & ?L) (EXISTS ?X ?S (?F & ?M))))
         ->
         (! $SUFFICE ?E
            (AND [IN ?X ?S]
                 & [! $DISTRIBUTEL (?XI ?YI) (?L ?M) (= ?XI ?YI)])))
Used in: ((DERIV2 : 30))

(RULE323 (SOLUTION-TEST->PATH-TEST USED 5 TIMES)
         •
         (?HSM WITH (SOLUTION-TEST : ?T)
               (PATH-TEST : ?R)
               (PATH : ?P)
               (CHOICES : ?C))
         ->
         (! $CONSIDER-PROP ?HSM PATH-TEST
            (AND ?R
                 [LAMBDA (?P)
                  (=> (=> (! $SUBST ?C ?P (?Q : (! $SELECT ?L))) ?Q) ?Q)]))
         IF
         ($NON-EMPTY! (?L : (! $SET-OFP (!! ?Q : (PREDICATE & ?ETC)) ?T T)))
         --
         (ADD CONSTRAINT ?Q FROM SOLUTION-TEST TO PATH-TEST WHEN IT SATISFIES
              MONOTONICITY CONDITION))
Used in: ((DERIV2 : 59) (DERIV13 : 20 44 65 92))

(RULE324 (SPECIAL-CASE-REDUCTION-RULE USED 1 TIMES)
         • (?E : (= (?H1 : ((?F : EXTREME) ?S1)) (?H2 : (?F ?S2))))
         -> (! $TRY ?E (SUBSET ?S2 ?S1) (IN ?H1 ?S2)))
Used in: ((DERIV2 : 63))

(RULE325 (SPECIAL-CASE-REDUCTION-RULE USED 1 TIMES)
         • (?E : (= (?S1 ?I) (?S2 ?I)))
         -> (! $TRY ?E (SUBSET ?S2 ?S1) (IN ?I (INDICES-OF ?S2))))
Used in: ((DERIV2 : 66))

(RULE326 (TRANSFER-RULE USED 1 TIMES)
         • (IN (?S ?I) ?S)
         -> (IN ?I (INDICES-OF ?S)))
Used in: ((DERIV2 : 69))

(RULE327 (PATH-TEST->STEP-TEST USED 3 TIMES)
         •
         (?HSM WITH (PATH-TEST : ?T)
               (STEP-TEST : ?S)
               (INDEX : ?I)
               (PATH : ?P))
         ->
         (! $TRY ?HSM
            (= (! $SELECT ?L) (FORALL ?I (INDICES-OF ?P)
                                      (?Q : (! Q*V Q))))
            (THEN $CONSIDER-PROP ?HSM STEP-TEST
                  (AND ?S
                       [LAMBDA  (?I (?C : (! Q*V (! $FILL-IN "choice variable"))
                                      ))
                        (THEN $SUBST ?C (?P ?I) ?Q])))
         IF
         ($NON-EMPTY! (?L : (! $SET-OFP (!! ?R : (PREDICATE & ?ETC)) ?T T))))
Used in: ((DERIV2 : 74) (DERIV13 : 28 72))

(RULE328 (TRANSFER-RULE USED 2 TIMES)
         • ((?Q : QUANTIFIER PREDICATE) ?X ?S ?E)
         ->
         (?Q (?I : (! Q*V (! $FILL-IN "quantifier variable")))
             (INDICES-OF ?S)
             (! $SUBST (?S ?I) ?X ?E)))
Used in: ((DERIV2 : 80) (DERIV13 : 73))

(RULE329 (PURE-TRANSPOSE-RULE USED 5 TIMES)
         • (?P : (PREDICATE & ?ETC))
         -> (?Q ?X ?S (! $SUBST ?E ?R ?P))
         IF ($INP! (!! ?R : ((?Q : QUANTIFIER PREDICATE)
                              ?X ?S ?E)) ?P)
         -- (SUBSUMES RULE220))
```

```
Used in: ((DERIV2 : 81)
          (DERIV9 : 32)
          (DERIV10 : 6 27)
          (DERIV12 : 6))

(RULE330 (TRANSFER-RULE USED 1 TIMES)
         * (?E : (FORALL ?I (?S : (INDICES-OF ?L)) ?P))
         -> (! $SUBST (NOT (AFTER ?X ?I)) ?R ?E)
         IF ($IMP! (!! ?R : (IN ?X ?S)) ?P)
         --
         (ASSUMING ?S IS A PREFIX OF SEQUENCE CONTAINING ?X -- FG HSM PATH))
Used in: ((DERIV2 : 82))

(RULE331 (CONTROL-RULE USED 3 TIMES)
         * (THEN ?F & ?L)
         -> (! APPLY# ?F ?L)
         --
         (THEN CONSTRUCT IS USED TO DELAY EVALUATION WHILE ARGUMENTS ARE PUT IN
               PROPER FORM))
Used in: ((DERIV2 : 89) (DERIV13 : 42 79))

(RULE332 (REDUCE-SEARCH-DEPTH USED 1 TIMES)
         *
         (THEN $CONSIDER-PROP
               (?HSM WITH (PATH : ?PATH)
                     (SEQUENCE : ?SEQUENCE)
                     (INDICES : ?INDICES)
                     (VARIABLES : ?VL)
                     (BINDINGS : ?BL))
               STEP-TEST ?R)
         ->
         (THEN $CONSIDER-PROP
               (?HSM WITH
                     (VARIABLES :
                                ((?V : (! Q*V (! $FILL-IN "variable name")))
                                 & ?VL))
                     (BINDINGS : ((?SEQUENCE ?X) & ?BL))
                     (INITIAL-PATH : (PROJECT ?SEQUENCE (PREFIX ?INDICES ?X))))
               STEP-TEST (! $SUBST ?V (?PATH ?X) ?R))
         IF ($IMP! (!! ?PATH ?X) ?R)
         -- (SEARCH STARTING AFTER ?X TO CHOOSE ?V))
Used in: ((DERIV2 : 93))

(RULE333 (HSM-CACHE USED 1 TIMES)
         *
         (THEN $CONSIDER-PROP
               (?HSM WITH (SEQUENCE : ?SEQUENCE)
                     (BINDINGS : ?BL)
                     (VARIABLES : ?VL))
               STEP-TEST ?R)
         ->
         (THEN $CONSIDER-PROP
               (?HSM WITH (BINDINGS : ((?E : (! $SELECT ?L))
                                        & ?BL))
                     (VARIABLES :
                                ((?V : (! Q*V (! $FILL-IN "variable name")))
                                 & ?VL)))
               STEP-TEST (! $SUBST ?V ?E ?R))
         IF
         ($NON-EMPTY! (?L :
                          (! $SET-OFP (!! ?E : (?F & ?ETC))
                             ?R (!! 1 $CAN-EXPAND-TO ?F ?SEQUENCE)))))
Used in: ((DERIV2 : 94))

(RULE334 (CONTROL-RULE USED 5 TIMES)
         * (THEN $CONSIDER-PROP ?HSM ?PROP ?V)
         -> (?HSM WITH (?PROP : ?V)))
Used in: ((DERIV2 : 95) (DERIV13 : 43 56 80 116))

(RULE335 (SOLUTION-TEST->PATH-ORDER USED 1 TIMES)
         * (?HSM WITH (SOLUTION-TEST : (LAMBDA (?PATH) (AND & ?L))))
         ->
         (! $TRY ?HSM
            (=> (! $SUBST (PREFIX ?PATH ANY) ?PATH (?P : (! $SELECT ?L))) ?P)
            (?HSM WITH (PATH-ORDER   (LAMBDA (?PATH) ?P)))))
Used in: ((DERIV2 : 96))

(RULE336 (CHECK-SUFF-COND-RULE USED 1 TIMES)
         * (?E : (=> ((?P : $DETECTOR!) ?S1) (?P ?S2)))
         -> (! $SUFFICE ?E (SUBSET ?S1 ?S2)))
Used in: ((DERIV2 : 97))

(RULE337 (OPPORTUNISTIC-EVALUATION-RULE USED 1 TIMES)
         * (SUBSET (PREFIX ?S ?I) ?S)
         -> T)
Used in: ((DERIV2 : 98))

(RULE338 (INTRODUCE-SPLIT-RULE PATH-ORDER->STEP-ORDER USED 1 TIMES)
         *
         (?HSM WITH (PATH-ORDER : (LAMBDA (?PATH) ?P))
               (STEP-TEST : (LAMBDA (?I ?C) ?ETC)))
         ->
         (! $CONSIDER-PROP ?HSM STEP-ORDER
            (LAMBDA (?I ?C) (! $SUBST (APPEND ?PATH (LIST ?C)) ?PATH ?P))))
Used in: ((DERIV2 : 101))

(RULE339 (ACTION-RULE USED 1 TIMES)
         * (= (CHOOSE ?C ?S (?ACT ?AGENT ?C)) (?ACT ?AGENT ?OBJ))
         -> (= ?S (SET ?OBJ))
         -- (?AGENT IS FORCED IF CHOICE SET IS SINGLETON))
Used in: ((DERIV3 : 7))

(RULE340 (INTRODUCE-SPLIT-RULE USED 2 TIMES)
         * (= ?S1 ?S2)
         -> (AND [(?R : (! $ORDERING-OF ?S1))
                 ?S1 ?S2] [?R ?S2 ?S1]}
         -- (PERMITS APPLICATION OF KNOWLEDGE ABOUT TRANSITIVE RELATIONS))
Used in: ((DERIV3 : 12) (DERIV13 : 83))

(RULE341 (SIMPLIFY->NON-CONSTANT-RULE USED 14 TIMES)
         * ((?R : COMM-CONN) (! $IDENTITY-OF ?R) ?P)
         -> ?P)
Used in: ((DERIV3 : 19)
          (DERIV5 : 9 25 34 43)
          (DERIV6 : 6 28 59)
          (DERIV9 : 48)
          (DERIV10 : 21)
          (DERIV11 : 13)
          (DERIV13 : 21 40 77))

(RULE342 (MAKE-ASSUMPTION-RULE USED 1 TIMES)
         * (= (?C : $CONSTANT!) ?E)
         -> (! $ASSUME ?E ?C))
Used in: ((DERIV3 : 26))

(RULE343 (SIMPLIFY->NON-CONSTANT-RULE USED 8 TIMES)
         * ((?R : COMM-CONN) ?P (! $IDENTITY-OF ?R))
         -> ?P)
Used in: ((DERIV3 : 36)
          (DERIV6 : 37 50)
          (DERIV8 : 13)
          (DERIV9 : 36 53)
          (DERIV14 : 26 78))

(RULE344 (INTRODUCE-SPLIT-RULE USED 1 TIMES)
         * (UNDO (?P : (= ?E ?V)))
         -> (AND ?P [BECOME ?E (! Q*V ?E)])
         -- (?E NEQ ?V BY DEFINITION OF BECOME))
Used in: ((DERIV3 : 49))

(RULE346 (USE-ASSUMPTION-RULE USED 9 TIMES)
         * ?E -> ?V IF ($IN! (?E ?V) (! $ASSUMPTIONS)))
Used in: ((DERIV3 : 62 - 63 ; 84)
          (DERIV4 : 17)
          (DERIV6 : 56)
          (DERIV8 : 12)
          (DERIV13 : 47 68 98))

(RULE347 (EXAMPLE-RULE USED 1 TIMES)
         * (?E : (EXISTS ?X ?S ?P))
         -> (! $SUFFICE ?E (! $SUBST (! $SELECT ?L) ?X ?P))
         IF ($NON-EMPTY! (?L : (! $SET-OFP (!! ?C : $CONSTANT!) ?P T))))
Used in: ((DERIV3 : 69))

(RULE349 (MAKE-ASSUMPTION-RULE USED 1 TIMES)
         * (=> ?P ?E)
         -> (! $ASSUMING ?P NIL T)
         -- (RULE OUT CASE))
Used in: ((DERIV3 : 80))

(RULE350 (REDUCE->NEC-COND-RULE USED 1 TIMES)
         * (ACHIEVE (AND & ?L))
         -> (ACHIEVE (AND & [! $REMOVE (! $SELECT ?L) ?L]))
         -- (REMOVE OVER-SPECIFIC GOAL CONJUNCT))
Used in: ((DERIV3 : 81))

(RULE351 (USE-ASSUMPTION-RULE USED 1 TIMES)
         * (ACHIEVE ?P)
         ->
         (ACHIEVE (AND & [! $DISTRIBUTE (!! ?E ?V) (! $ASSUMPTIONS) (= ?E ?V)]))
         -- (HACK))
Used in: ((DERIV3 : 88))

(RULE352 (RECOGNIZE-RULE USED 1 TIMES) * (= ?P NIL) -> (NOT ?P))
Used in: ((DERIV3 : 89))

(RULE353 (ACTION-RULE USED 1 TIMES)
         * (NOT (= ?E ?P))
         -> (= ?E ME)
         --
         (MAKE YOURSELF ?E TO PREVENT ?P FROM DOING IT -- ASSUMING ?P NEQ ME))
Used in: ((DERIV3 : 91))

(RULE354 (USE-ASSUMPTION-RULE REDUCE->NEC-COND-RULE USED 2 TIMES)
         * (?E : (AND & ?L))
         -> (! $SUBST (! $SUBST ?C ?V ?Q) ?Q ?E)
         IF
         (! $EXISTS! (!! ?P : (= ?V ?C))
            ?L (!! ! $EXISTS! (!! ?Q) (! $REMOVE ?P ?L) (!! ! $INP! ?V ?Q))))
Used in: ((DERIV3 : 95) (DERIV5 : 32))

(RULE356 (INTRODUCE-SPLIT-RULE HISTORY-RULE USED 1 TIMES)
         * (?P : (AT ?OBJ ?LOC))
         ->
         (OR [WAS-DURING (CURRENT (?A : (! $SELECT (! $ACTIONS))))
                         (CAUSE ?P)]
             [BEFORE (CURRENT ?A) ?P]))
Used in: ((DERIV4 : 47))

(RULE357 (FACT-RULE USED 1 TIMES)
         * (BEFORE (CURRENT ROUND-IN-PROGRESS)
                   (AT (?C : CARD) (PILE ?P)))
         -> NIL)
Used in: ((DERIV4 : 48))

(RULE358 (RECOGNIZE-RULE ACTION-RULE USED 3 TIMES)
         * (CAUSE (AT ?OBJ ?LOC))
         -> (MOVE ?OBJ (! Q*V LOC) ?LOC))
Used in: ((DERIV4 : 51) (DERIV10 : 8) (DERIV12 : 9))

(RULE359 (PURE-TRANSPOSE-RULE USED 1 TIMES)
         * (AND [UNTIL ?P ?A] & ?L)
         -> (UNTIL ?P (AND ?A & ?L)))
Used in: ((DERIV5 : 3))

(RULE360 (PURE-TRANSPOSE-RULE PROBLEM-MERGING-RULE USED 1 TIMES)
          * (AND & ?L)
          -> (ACHIEVE (AND & [! $DISTRIBUTE (!! ACHIEVE ?P) ?L ?P]))
          IF (NOT (! $IN! (!! ~ (ACHIEVE ?P)) ?L)))
Used in: ((DERIV5 : 4))

(RULE361 (PROPAGATE-SPLIT-RULE USED 2 TIMES)
          * (IN ?C (FILTER ?S ?P))
          -> (AND [IN ?C ?S] [?P ?C]))
Used in: ((DERIV5 : 41 41))

(RULE362 (SPECIAL-CASE-REDUCTION-RULE USED 1 TIMES)
          * (?E : ((?Q : QUANTIFIER PREDICATE)
                   ?X ?S ?P))
          -> (! $TRY ?E (SUBSET ?S ?SS) (?Q ?X ?S (! $REMOVE ?R ?P)))
          IF (! $IN! (!! ?R : (IN ?X ?SS)) ?P))
Used in: ((DERIV6 : 17))

(RULE363 (FACT-RULE USED 1 TIMES)
          * (SUBSET (CARDS-PLAYED) (CARDS))
          -> T)
Used in: ((DERIV6 : 18))

(RULE364 (USE-ASSUMPTION-RULE EXAMPLE-RULE USED 1 TIMES)
          * (?E : (IN (?C : $UNBOUND-VAR!) (PROJECT ?F ?S)))
          -> (! $TRY ?E (IN ?X ?S) (! $BIND-VAR ?C ?V))
          IF ($IN! (!! (?F ?X) ?V) (! $ASSUMPTIONS)))
Used in: ((DERIV6 : 47))

(RULE365 (FACT-RULE USED 1 TIMES) * (IN (LEADER) (PLAYERS)) -> T)
Used in: ((DERIV6 : 48))

(RULE366 (TEMPORAL-GOAL-RULE REDUCE->NON-EQUIVALENT-RULE USED 1 TIMES)
          * (ACHIEVE ?P)
          -> (ACHIEVE (! $SUBST ?X ?E ?P))
          IF ($INP! (!! ?E : (PREVIOUS ?X)) ?P))
Used in: ((DERIV7 : 3))

(RULE367 (RECOGNIZE-RULE USED 1 TIMES)
          * (NOT (?P ?X))
          -> ((! $OPPOSITE-OF ?P) ?X))
Used in: ((DERIV7 : 8))

(RULE368 (RECOGNIZE-RULE ACTION-RULE USED 2 TIMES)
          * (UNDO (AT ?OBJ ?LOC))
          -> (MOVE ?OBJ ?LOC (! Q*V LOC)))
Used in: ((DERIV8 : 10) (DERIV10 : 29))

(RULE369 (REDUCE->NEC-COND-RULE USED 1 TIMES)
          * (IN ?C (SET-OF ?X ?S ?P))
          -> (! $SUBST ?C ?X ?P)
          -- (LIKE RULE10 BUT ASSUME (IN ?C ?S)))
Used in: ((DERIV9 : 30))

(RULE370 (INTRODUCE-SPLIT-RULE HISTORY-RULE USED 1 TIMES)
          * (?E : (# (SET-OF ?X ?S ?P)))
          ->
          (! $TRY ?E
             (NOT (DURING (! $INSTANTIATE (?A : (! $SELECT (! $ACTIONS))))
                          (CAUSE ?P)))
             (- (# (SET-OF ?X ?S (BEFORE (CURRENT ?A) ?P)))
                (# (SET-OF ?X ?S (WAS-DURING (CURRENT ?A) (UNDO ?P))))))
          -- (BASIS FOR CACHE-AND-UPDATE SCHEME))
Used in: ((DERIV10 : 4))

(RULE371 (PURE-TRANSPOSE-RULE PROPAGATE-SPLIT-RULE USED 2 TIMES)
          * (BEFORE ?A (?P & ?L))
          -> (?P & (! $DISTRIBUTE ?X ?L (BEFORE ?A ?X))))
Used in: ((DERIV10 : 15 - 16))

(RULE372 (FACT-RULE USED 1 TIMES)
          * (BEFORE (CURRENT ROUND-IN-PROGRESS) (IN-POT ?C))
          -> NIL)
```

```
Used in: ((DERIV10 : 17))

(RULE373 (FACT-RULE USED 1 TIMES) • (AT ?C HOLE) -> NIL)
Used in: ((DERIV10 : 19))

(RULE374 (HEURISTIC-EQUIVALENCE-PRESERVING-RULE USED 1 TIMES)
         • (BEFORE ?A (?P : BOOLEAN $CONSTANT!))
         -> ?P)
Used in: ((DERIV10 : 20))

(RULE375 (PROPAGATE-SPLIT-RULE USED 1 TIMES)
         • (# (SET-OF ?X ?S (NOT ?P)))
         -> (- (# ?S) (# (SET-OF ?X ?S ?P))))
Used in: ((DERIV10 : 23))

(RULE376 (FACT-RULE USED 1 TIMES) • (# (CARDS-IN-SUIT ?S)) -> 13)
Used in: ((DERIV10 : 24))

(RULE377 (RECOGNIZE-RULE USED 1 TIMES)
         • (UNDO (NOT ?P))
         -> (CAUSE ?P))
Used in: ((DERIV12 : 3))

(RULE378 (SOLUTION-TEST->COMPLETION-TEST USED 1 TIMES)
         • (?HSM WITH (SOLUTION-TEST : (LAMBDA (?S) (AND & ?L))))
         ->
         (?HSM WITH (COMPLETION-TEST : (LAMBDA (?S) ?P))
              (SOLUTION-TEST : (LAMBDA (?S) (AND & [! $REMOVE ?P ?L]))))
         IF ($EXISTS! ?P ?L (!! ! $INP! (!! # ?S) ?P))
         -- (?P SHOULD NOT DEPEND OTHERWISE ON ?S)
         (MOVE CONSTRAINT ON SOLUTION LENGTH TO COMPLETION TEST))
Used in: ((DERIV13 : 19))

(RULE379 (PARTIAL-MATCH-RULE USED 2 TIMES)
         •
         (?E :
            (=> (FORALL ?X ?S1 ?P) (FORALL ?Y ?S2 (! $TEST-SUBST ?Y ?X ?P))))
         -> (! $SUFFICE ?E (SUBSET ?S2 ?S1)))
Used in: ((DERIV13 : 22 67))

(RULE380 (PARTIAL-MATCH-RULE USED 1 TIMES)
         • (= (FORALL ?X ?S1 ?P) (FORALL ?Y ?S2 ?Q))
         -> (= ?Q (OR [! $SUBST ?Y ?X ?P] [IN ?Y (DIFF ?S2 ?S1)]))
         -- (ASSUMING (SUBSET ?S1 ?S2)))
Used in: ((DERIV13 : 31))

(RULE381 (ELABORATE-RULE USED 1 TIMES)
         • (INDICES-OF ?S)
         -> (LB:UB 1 (# ?S))
         -- (ASSUMING ?S INDEXED BY INTEGER))
Used in: ((DERIV13 : 32))

(RULE382 (RECOGNIZE-RULE USED 1 TIMES)
         • (DIFF (LB:UB ?L1 ?U) (LB:UB ?L2 ?U))
         -> (LB:UB ?L1 (- ?L2 1))
         -- (SIMILAR TO RULE252))
Used in: ((DERIV13 : 33))

(RULE383 (CHECK-SUFF-COND-RULE USED 2 TIMES)
         • (?E : (=> ((?R : TRANSITIVE) ?X ?Z) (?R ?Y ?Z)))
         -> (! $SUFFICE ?E (?R ?Y ?X)))
Used in: ((DERIV13 : 45 93))

(RULE384 (STEP-TEST->GENERATOR USED 1 TIMES)
         •
         (?HSM WITH (CHOICE-SETS : (LAMBDA (?I) ?S))
              (STEP-TEST : (LAMBDA (?I ?C) ?T)))
         ->
         (THEN $CONSIDER-PROP (?HSM WITH (STEP-TEST : T))
              CHOICE-SETS (LAMBDA (?I) (SET-OF ?C ?S ?T)))
         -- (INCORPORATE TEST INTO GENERATOR))
Used in: ((DERIV13 : 52))
```

```
(RULE385 (INTRODUCE-SPLIT-RULE USED 1 TIMES)
         * (SET-OF ?X ?S (OR & ?L))
         -> (IF ?P ?S (SET-OF ?X ?S (OR & [! $REMOVE ?P ?L])))
         IF ($EXISTS! ?P ?L (!! ! $INDEPENDENT-OF ?P ?X)))
Used in: ((DERIV13 : 53))

(RULE386 (GENERATOR->PRECOMPUTE USED 1 TIMES)
         *
         (THEN $CONSIDER-PROP (?HSM WITH (PATH : ?PATH)) CHOICE-SETS ?CS)
         ->
         (THEN $CONSIDER-PROP ?HSM CHOICE-SETS
               (! $SUBST
                  ((! $DEFINE (?F : (! $FILL-IN "Function name"))
                      ((?Y : (! Q*V ?X)))
                      (! $SUBST ?Y (?C : (! $SELECT ?L)) ?FS))
                   ?C)
                  ?FS ?CS))
         IF
         ($EXISTSP! (!! ?FS : (SET-OF ?X ?S ?P))
                    ?CS
                    (!! ! $NON-EMPTY!
                        (?L : (! $SET-OFP (!! ?C) ?P (!! ! $INP! ?PATH ?C)))))
         --
         (FUNCTION ?F CAN BE PRECOMPUTED IF ?S IS SMALL AND ?FS DEPENDS ON ?PATH
                   ONLY THROUGH ?C))
Used in: ((DERIV13 : 55))

(RULE387 (RECOGNIZE-RULE USED 1 TIMES)
         * (>= (# (SET-OF ?X ?S ?P)) 1)
         -> (EXISTS ?X ?S ?P)
         -- (CONVERSE OF RULE291))
Used in: ((DERIV13 : 85))

(RULE388 (PARTIAL-MATCH-RULE USED 1 TIMES)
         *
         (?E :
             (SUBSET (SET-OF ?X ?S1 ?P)
                     (SET-OF ?Y ?S2 (! $TEST-SUBST ?Y ?X ?P))))
         -> (! $SUFFICE ?E (SUBSET ?S1 ?S2))
         -- (LIKE RULE30 WITH (?F ?S) = (SET-OF ?X ?S ?P))
         (NOTE SIMILARITY TO RULE379))
Used in: ((DERIV13 : 97))

(RULE389 (INTRODUCE-SPLIT-RULE PATH-TEST->STEP-TEST USED 1 TIMES)
         *
         (?HSM WITH (PATH-TEST : ?PT)
               (PATH : ?PATH)
               (STEP-TEST : (LAMBDA (?I ?C) ?ST)))
         ->
         (THEN $CONSIDER-PROP ?HSM STEP-TEST
               (LAMBDA (?I ?C)
                (AND ?ST
                     [! $SUBST (APPEND ?PATH (LIST ?C)) ?PATH (! $SELECT ?L)])))
         IF ($NON-EMPTY! (?L : (! $SET-OFP (!! PREDICATE & ?ETC) ?PT T)))
         -- (SIMILAR TO RULE338))
Used in: ((DERIV13 : 105))

(RULE390 (SIMPLIFY-BY-INDUCTION USED 1 TIMES)
         *
         (?E :
             (THEN $CONSIDER-PROP (?HSM WITH (PATH-TEST : ?PT)
                                             (PATH : ?PATH))
                   STEP-CONSTRAINT ?ST))
         -> (! $SUBST T ?P ?E)
         IF
         ($IN! (LAMBDA (?PATH)
                (?P : (! $SELECT (! $SET-OFP (!! PREDICATE & ?ETC) ?ST T))))
               ?PT)
         -- (?P IS TRUE FOR EXISTING PATH BY INDUCTION))
Used in: ((DERIV13 : 108))
```

```
(RULE391 (RECOGNIZE-RULE USED 1 TIMES)
         * (IF ?P ?X T)
         -> (OR [NOT ?P] ?X)
         -- (SIMILAR TO RULE294))
Used in: ((DERIV13 : 109))

(RULE392 (RECOGNIZE-RULE USED 1 TIMES)
         * (=< (?E : $NON-NEGATIVE!) 0)
         -> (= ?E 0))
Used in: ((DERIV13 : 111))

(RULE393 (PROPAGATE-SPLIT-RULE FUNCTIONALIZE-RULE USED 2 TIMES)
         * (LAMBDA (?I) (?F & ?L))
         -> (?F & (! $DISTRIBUTE (!! ?X) ?L (!! (LAMBDA (?I) ?X))))
         -- (TREAT ?F AS FUNCTIONAL))
Used in: ((DERIV14 : 13 16))
NIL
<90>recordfile

RECORD FILE DSK: (RUL319 . QZG) CLOSED 20-MAR-81 18:35:08
```

Appendix C
Conceptual Knowledge

This appendix includes FOO's is-a hierarchy (§C.1) and semantic relations (§C.2), and lists the definitions of concepts used in task descriptions and advice (§C.3).

C.1. Is-a Hierarchy

FOO's is-a hierarchy provides connections between general concepts used in transformation rules and specific concepts used in task advice. The hierarchy is tangled: a concept may have more than one immediate generalization. For example, NON-NEGATIVE-INTEGER is both NON-NEGATIVE and an INTEGER.

The semantics of the "is-a" relationship are deliberately left vague. Roughly speaking, the non-terminal nodes represent concept features (*e.g.*, REFLEXIVE, HEARTS), and the is-a hierarchy marks concepts in such a way that a concept with a given marking, *e.g.*, REFLEXIVE, inherits more generalized markings, *e.g.*, PREDICATE, FUNCTION. Each feature can be viewed as implicitly specifying a particular "is-a" relationship; for example, the marking HEARTS on the concept PLAY can be interpreted as meaning "PLAY is-a *concept relevant to the domain of* HEARTS."

The is-a hierarchy includes both constants and functions. FOO has definitions, listed in Appendix C.3, for some of the concepts, *e.g.*, PLAY, but not for others, *e.g.*, DURING.

```
RECORD FILE DSK: (ISA319 . QZG) OPENED 20-MAR-81 18:15:09
<70>p-f any-concept

 ANY-CONCEPT:
      AFFECT:  (CAUSE UNDOES)
           CHANGE:  (CAUSE REMOVE-1-FROM UNDO)
      BOOLEAN:  T
           NIL:  0
      CARD:  QS
      COLLECTION:
           LIST:  SET
      COMM-CONN:
           CONJ:  (AND D* INTERSECT)
           JOIN:  (APPEND SCENARIO)
                DISJ:  (+ D+ OR UNION)
      COMPUTABLE:  (+ - AND NEQ NOT OR)
           DEPENDENCE-OP:  (D* D+ D-)
      CONSTANT:  (1 T)
           NIL:  0
      CONSTRUCTED:  CANTUS
      FILTERED:  FORALL
           EXISTENTIAL:  (SET-OF THE)
                EXISTS:  EXISTS-UNIQUE
      FORMULA:  PR-DISJOINT-FORMULA
      FUNCTION:
           FXTREME:  (HIGHEST HIGHEST-IN-SUIT-LED LOWEST)
           HOMOMORPHISM:  (#OCCURRENCES CHOICE-SEQ-OF DURING HAVE-POINTS IN SET-OF)
           INJECTIVE:  (CARD-OF EACH PLAY PLAY-CARD TAKE)
                DISTRIBUTE:  EACH
           PREDICATE:  (AT BREAKING-SUIT CAN DISJOINT EXTEND-TONE FLUSH FOLLOWING FORALL HAS
                          HAS-ME HAVE-POINTS IN IN-POT IRREVERSIBLE-DURING LEAD-SAFE-SPADE
                          LEAD-SPADE LEAD-TO LEADING LEGAL LEGAL-CANTUS! LOW MUST NON-VOID
                          OUT OVERLAP PLAY-SPADE POSSIBLE QS-OUT ROUND-IN-PROGRESS
                          SAFE-SPADE! SPADE! SPLIT TAKEN TRICK-HAS-POINTS TRICK-IN-PROGRESS
                          UNCHANGEABLE UNDOABLE-DURING USABLE-INTERVAL! VOID)
                ACTION:  (DEAL EXTEND MOVE PLAY PLAY-CARD ROUND-IN-PROGRESS TAKE TAKE-TRICK
                              TRICK)
                ANTI-SYMMETRIC:  (HIGHER LOWER)
                BOOLEAN-OP:  (=> AND NOT OR)
                EXISTS:  EXISTS-UNIQUE
                REFLEXIVE:  (=> DURING SUBSET)
                     EQUIVALENCE:  =
                TRANSITIVE:  (=> SUBSET)
                     COMPARATIVE:  (=< >= HIGHER LOWER)
                     EQUIVALENCE:  =
           QUANTIFIER:  (CHOOSE EACH FOR-SOME FORALL SET-OF THE)
                DISTRIBUTE:  EACH
                EXISTS:  EXISTS-UNIQUE
           SET-FUNCTION:  (CARDS CARDS-IN-HAND PROJECT SET SET-OF)
                DISTRIBUTE:  EACH
      HEARTS:  (DEAL PLAY PLAY-CARD ROUND-IN-PROGRESS TAKE TAKE-TRICK TRICK)
      INCR1:  #CHOOSE
           INCR1-DECR2:  (- //)
      MAPPED:  (CHOOSE EACH FOR-SOME)
           DISTRIBUTE:  EACH
      MUSIC:  EXTEND
      NUMBER:
           INTEGER:
                NON-NEGATIVE-INTEGER:  (0 1 #)
           NON-NEGATIVE:
                NON-NEGATIVE-INTEGER:  (0 1 #)
      NUMERICAL:  (= =< >=)
      ONE-OF:
           EXTREME:  (HIGHEST HIGHEST-IN-SUIT-LED LOWEST)
      SEQUENCE:  CANTUS
      STEP-CONSTRAINT:  (STEP-ORDER STEP-TEST)
      UNTYPED:  (CHOOSE EACH SET-OF THE WHILE)
           DISTRIBUTE:  EACH
<71>recordfile
RECORD FILE DSK: (ISA319 . QZG) CLOSED 20-MAR-81 18:22:31
```

C.2. Semantic Relations

Semantic relationships among concepts in FOO are represented as triples of the form `<attribute> of <object> is <value>`:

```
RECORD FILE DSK: (SEM319 . QZG) OPENED 19-MAR-81 21:47:17
NIL
<98>p-relations


    ADD of #OCCURRENCES is +
    ADD of CHOICE-SEQ-OF is APPEND
    ADD of DURING is OR
    ADD of HAVE-POINTS is OR
    ADD of IN is OR
    ADD of SET-OF is UNION

    COMPARATIVE of HIGHEST is HIGHER
    COMPARATIVE of LOWEST is LOWER

    IDENTITY of + is 0
    IDENTITY of AND is T
    IDENTITY of D* is INCREASING

    OPPOSITE of =< is >=
    OPPOSITE of >= is =<
    OPPOSITE of HIGH is LOW
    OPPOSITE of HIGHER is LOWER
    OPPOSITE of LOW is HIGH
    OPPOSITE of LOWER is HIGHER

    PREDICATE of HIGHER is HIGH
    PREDICATE of LOWER is LOW

    SUPERLATIVE of HIGHER is HIGHEST
    SUPERLATIVE of LOWER is LOWEST
NIL
<99>recordfile

RECORD FILE DSK: (SEM319 . QZG) CLOSED 19-MAR-81 21:47:30
```

C.3. Concept Definitions

Concept definitions are represented in FOO as LISP function definitions. The somewhat awkward construct `(?X : (! Q*V X))` is used to generate a unique variable name each time the definition containing it is instantiated, so as to prevent name conflicts in expressions with nested quantifiers. The generated name consists of `X` followed by a number: `X1, X2,` ....

```
RECORD FILE DSK: (DEF319 . QZG) OPENED 19-MAR-81 23:23:34
NIL
<60>(length defined-fns)

112
<61>p-definitions

(DE #CARDS-IN-HAND (P) (# (CARDS-IN-HAND P)))

(DE #CARDS-OUT NIL (# (CARDS-OUT)))

(DE #CARDS-OUT-IN-SUIT (SUIT) (# (CARDS-OUT-IN-SUIT SUIT)))

(DE #OCCURRENCES (N C) (# (SET-OF (?X : (! Q*V X)) C (= ?X N))))

(DE AT (C W) (= (LOC C) W))

(DE AVOID (E S) (ACHIEVE (NOT (DURING S E))))

(DE AVOID-TAKING-POINTS (TRICK) (AVOID (TAKE-POINTS ME) TRICK))

(DE BECOME (F C)
    (LAMBDA (I) (AND [NOT (= (F I) (C I))] [= (F (+ I 1)) (C I)])))

(DE BREAK-SUIT (P)
    (WHILE (NOT (IN-SUIT (CARD-OF P) (SUIT-LED))) (PLAY-CARD P)))

(DE BREAKING-SUIT (P) (NOT (IN-SUIT (CARD-OF P) (SUIT-LED))))

(DE CAN (A) (EXISTS (?X : (! Q*V A)) (ACTIONS) (A ?X)))

(DE CANTUS NIL
    (EACH (?I : (! Q*V I))
          (LB:UB 1 (CANTUS-LENGTH))
          (CHOOSE-NOTE ?I)))

(DE CANTUS! (C) (EACH I (LB:UB 1 (# C)) (NOTE! I)))

(DE CANTUS-1 (J) (PREFIX CANTUS J))

(DE CARD-OF (P)
    (THE (?C : (! Q*V C))
         (CARDS-PLAYED)
         (DURING (CURRENT TRICK) (PLAY P ?C))))

(DE CARDS-IN-HAND (P) (SET-OF (?C : (! Q*V C)) (CARDS) (HAS P ?C)))

(DE CARDS-IN-POT NIL (SET-OF (?C : (! Q*V C)) (CARDS) (AT ?C POT)))

(DE CARDS-IN-SUIT (SUIT)
    (SET-OF (?C : (! Q*V C)) (CARDS) (IN-SUIT ?C SUIT)))

(DE CARDS-IN-SUIT-LED (S) (FILTER S IN-SUIT-LED))

(DE CARDS-OUT NIL (FILTER (CARDS) OUT))

(DE CARDS-OUT-IN-SUIT (SUIT) (FILTER (CARDS-IN-SUIT SUIT) OUT))

(DE CARDS-PLAYED NIL (PROJECT CARD-OF (PLAYERS)))
```

```
(DE CAUSE (P) (LAMBDA (I) (AND [NOT (P I)] [P (+ I 1)])))

(DE CHOOSE-NOTE (I) (CHOOSE (NOTE I) (TONES) (NOTE I)))

(DE CLIMAX (C) (HIGHEST C))

(DE DEAL (C P) (GET-CARD P C DECK))

(DE DISJOINT (S1 S2) (NOT (OVERLAP S1 S2)))

(DE EXCEPT (C S) (SET-OF (?X : (! Q*V X)) S (NOT (= ?X C))))

(DE EXISTS-UNIQUE (X S P) (= (# (SET-OF X S P)) 1))

(DE EXTEND (E S) (BECOME S (APPEND S (LIST E))))

(DE EXTEND-TONE (N C)
    (AND [EXTEND (?X : (! Q*V X)) C] [= (TONE-OF ?X) N]))

(DE FILTER (S P) (SET-OF (?X : (! Q*V X)) S (P ?X)))

(DE FLUSH (C) (MUST (OWNER-OF C) (PLAY (OWNER-OF C) C)))

(DE FOLLOWING (P) (NOT (LEADING P)))

(DE GET-CARD (P C S) (MOVE C S (HAND P)))

(DE HANDS (S) (PROJECT HAND S))

(DE HAS (P C) (AT C (HAND P)))

(DE HAS-ME (C) (HAS ME C))

(DE HAS-POINTS (C) (> (POINTS C) 0))

(DE HAVE-POINTS (S) (EXISTS (?C : (! Q*V C)) S (HAS-POINTS ?C)))

(DE HEART! (C) (= (SUIT-OF C) H))

(DE HIGHEST (S)
    (THE (?C : (! Q*V C))
         S (FORALL (?X : (! Q*V X)) S (NOT (HIGHER ?X ?C)))))

(DE HIGHEST-IN-SUIT-LED (S) (HIGHEST (CARDS-IN-SUIT-LED S)))

(DE IN-POT (C) (AT C POT))

(DE IN-SUIT (C SUIT) (= (SUIT-OF C) SUIT))

(DE IN-SUIT-LED (C) (= (SUIT-OF C) (SUIT-LED)))

(DE INTERVALS-OF (S)
    (DISTRIBUTE (?I : (! Q*V I))
                (LB:UB 2 (# S))
                (INTERVAL (S (- ?I 1)) (S ?I))))

(DE IRREVERSIBLE-DURING (S P) (NOT (UNDOABLE-DURING S P)))

(DE LB:UB (LB UB)
    (SET-OF (?K : (! Q*V K)) (INTEGERS) (BETWEEN ?K LB UB)))

(DE LEAD (P C) (WHILE (LEADING P) (PLAY P C)))

(DE LEAD-CARD (P) (WHILE (LEADING P) (PLAY-CARD P)))

(DE LEAD-SAFE-SPADE (P)
    (AND [LEADING P] [SPADE! (CARD-OF P)] [HIGHER QS (CARD-OF P)]))

(DE LEAD-SPADE (P) (AND [LEADING P] [SPADE! (CARD-OF P)]))

(DE LEAD-TO (EVENT1 EVENT2 TIME)
    (=> (DURING TIME EVENT1) (DURING TIME EVENT2)))

(DE LEADER NIL (PREVIOUS (TRICK-WINNER)))

(DE LEADING (?P) (= (LEADER) ?P))

(DE LEGAL (P C)
    (AND [HAS P C]
         [=> (LEADING P) (OR [CAN-LEAD-HEARTS P] [NEQ (SUIT-OF C) H])]
         [=> (FOLLOWING P)
             (OR [VOID P (SUIT-LED)] [IN-SUIT C (SUIT-LED)])]))

(DE LEGAL-CANTUS! (S)
    (AND [IN (# S) (LB:UB 8 16)]
         [FORALL (?I : (! Q*V INTERVAL))
                 (INTERVALS-OF S)
                 (USABLE-INTERVAL! ?I)]
         [SUBSET (MELODIC-RANGE S) (MAJOR 10)]
         [= (#OCCURRENCES (CLIMAX S) S) 1]))

(DE LEGALCARDS (P) (SET-OF (?C : (! Q*V C)) (CARDS) (LEGAL P ?C)))

(DE LOWEST (S)
    (THE (?C : (! Q*V C))
         S (FORALL (?X : (! Q*V X)) S (NOT (LOWER ?C ?X)))))

(DE MELODIC-RANGE (M) (SIZE (INTERVAL (LOWEST M) (HIGHEST M))))

(DE MOVE (OBJ SOURCE DEST)
    (AND [= (LOC OBJ) SOURCE] [BECOME (LOC OBJ) DEST]))

(DE MUST (AGT ACT) (= (ACTIONS-OF AGT) ACT))

(DE NEXT (E) (LAMBDA (J) (E (+ J 1))))

(DE NEXT-LEADER NIL (TRICK-WINNER))

(DE NEXT-NOTE NIL (CHOOSE (?N : (! Q*V)) (TONES) ?N))

(DE NON-VOID (P) (NOT (VOID P (SUIT-LED))))

(DE NOTE (I) (NTH CANTUS I))

(DE NOTE! (I) (LAMBDA (N) (= N (NOTE I))))

(DE OCCURS-ONCE-BY! (TONE I)
    (= (#OCCURRENCES TONE (PREFIX CANTUS I)) 1))

(DE OPPONENTS (P) (DIFF (PLAYERS) (SET P)))

(DE OUT (C) (EXISTS (?P : (! Q*V P)) (OPPONENTS ME) (HAS ?P C)))

(DE OVERLAP (S1 S2) (EXISTS (?X : (! Q*V X)) S1 (IN ?X S2)))

(DE OWNER-OF (C) (THE P (PLAYERS) (HAS P C)))

(DE PASS (P1 C P2) (GET-CARD P2 C (HAND P1)))

(DE PILES (S) (PROJECT PILE S))

(DE PLAY (P C) (MOVE C (HAND P) POT))

(DE PLAY-CARD (P)
    (CHOOSE (CARD-OF P) (LEGALCARDS P) (PLAY P (CARD-OF P))))

(DE PLAY-SPADE (P) (AND [PLAY P (?C : (! Q*V C))] [IN-SUIT ?C S]))

[DE PLAY-SUIT (P SUIT)
    (AND [PLAY P (?C : (! Q*V C))] [IN-SUIT ?C SUIT])]

[DE PLAYER-OF (C)
    (THE (?P : (! Q*V P)) (PLAYERS) (= (CARD-OF ?P) C))]
```

```
(DE POINT-CARDS NIL
    (SET-OF (?C : (! Q*V C)) (CARDS) (HAS-POINTS ?C)))

(DE POINTS (C)
    (IF (= C QS) 13 (IF (HEART! C) 1 (IF (= C JD) -10 0))))

(DE PR-DISJOINT (H S U) (PR-DISJOINT-FORMULA (# H) (# S) (# U)))

(DE PR-DISJOINT-FORMULA (#H #S #U)
    (// (#CHOOSE (- #U #S) #H) (#CHOOSE #U #H)))

(DE PROJECT (F S) (EACH (?X : (! Q*V X)) S (F ?X)))

(DE QS-OUT NIL (OUT QS))

(DE QSO NIL (OWNER-OF QS))

(DE ROUND-IN-PROGRESS NIL (EACH I (TRICK-INDICES) (TRICK)))

(DE ROUND-PROGRESSING NIL T)

(DE SAFE-SPADE! (C) (AND [SPADE! C] [LOWER C QS]))

[DE SPADE! (C) (IN-SUIT C S)]

(DE SPADES-IN (L) (SET-OF (?C : (! Q*V C)) L (= (SUIT-OF ?C) S)))

(DE SPLIT (B P) (OR [AND B [=> B P]] [AND [NOT B] [=> (NOT B) P]]))

(DE SUIT-LED NIL (SUIT-OF (CARD-OF (LEADER))))

(DE TAKE (P C) (MOVE C POT (PILE P)))

(DE TAKE-POINTS (P)
    (FOR-SOME (?C : (! Q*V C)) (POINT-CARDS) (TAKE P ?C)))

(DE TAKE-TRICK (P)
    (EACH (?C : (! Q*V C)) (CARDS-PLAYED) (TAKE P ?C)))

(DE TAKEN (C) (EXISTS P (PLAYERS) (TOOK P C)))

(DE TONES-OF (C) (PROJECT TONE-OF C))

(DE TOOK (P C) (WAS-DURING (CURRENT ROUND-IN-PROGRESS) (TAKE P C)))

(DE TRICK NIL
    (SCENARIO (EACH (?P : (! Q*V P)) (PLAYERS) (PLAY-CARD ?P))
              (TAKE-TRICK (TRICK-WINNER))))

(DE TRICK-HAS-POINTS NIL (HAVE-POINTS (CARDS-PLAYED)))

(DE TRICK-IN-PROGRESS NIL T)

(DE TRICK-WINNER NIL (PLAYER-OF (WINNING-CARD)))

(DE UNCHANGEABLE (P) (NOT (CAN (CHANGE P))))

(DE UNDO (P) (LAMBDA (I) (AND [P I] [NOT (P (+ I 1))])))

(DE UNDOABLE-DURING (S Q)
    (EXISTS (?ACT : (! Q*V ACT))
            (ACTIONS)
            (AND [DURING S ?ACT] [UNDOES ?ACT Q])))

[DE UNDOES (A Q) (CAUSE A (NOT Q))]

(DE USABLE-SUCCESSORS-OF (NOTE2)
    (SET-OF NOTE1 (TONES) (USABLE-INTERVAL! (INTERVAL NOTE2 NOTE1))))

(DE VOID (P SUIT)
    (NOT (EXISTS (?C : (! Q*V C)) (CARDS-IN-HAND P) (IN-SUIT ?C SUIT))))
```

```
(DE WINNING-CARD NIL (HIGHEST-IN-SUIT-LED (CARDS-PLAYED)))

<62>recordfile

RECORD FILE DSK: (DEF319 . QZG) CLOSED 19-MAR-81 23:26:51
```

Appendix D
Derivations

This appendix describes the notation used in the derivations and presents each derivation in its entirety.

D.1. Notation Used in Derivations

The notation for the nth step of DERIVd is as follows:

```
                    <expression>
|||  d:n            --- [<ruletype> by RULEk] --->
                    <expression>
<var> <- <value>
<schema> <- <slot> : <value>
<expression> <- ; <annotation>
```

The first expression is a sub-expression of the current overall expression. Step n of derivation d applies RULEk to this sub-expression, producing the second expression, which is substituted for it. The "type" of RULEk is intended to suggest the nature of the transformation, but otherwise has no significance.

The three vertical bars indicate that the current expression is part of a subgoal three deep on the subgoal stack. If this step were initiating this subgoal, *i.e.*, pushing it onto the stack, it would be indicated by

```
||>  d:n            --- [<ruletype> by RULEk] --->
```

Similarly, if it were satisfying the subgoal, *i.e.*, popping it off the stack, it would be indicated by

```
||<  d:n            --- [<ruletype> by RULEk] --->
```

Besides transforming a sub-expression of the current expression, a step may generate some global information (§5.4), such as an assumption, a variable binding, a slot value for a schema, or an annotation on an expression. Such global information is indicated by notation following the new expression. For example, the act of binding a value e to a variable v is indicated as

```
v <- BINDING : e
```

Filling in e as the value of slot p in schema s is indicated as

```
s <- p : e
```

Adding an annotation to an expression is indicated as

```
e <- ; <annotation>
```

This is illustrated at step 2:39:

```
((HAS P1 C2) <- ; (=> (OUT C2) IF (IN P1 (OPPONENTS ME))))
```

The act of assuming that an expression e has value v is indicated rather clumsily as an annotation:

```
e <- ; (-> v if (= e v))
```

D.2. DERIV2: Heuristic Search to Avoid Taking Points

```
RECORD FILE DSK: (DV2D07 . PZG) OPENED 07-DEC-80 17:59:40
NIL
<71>p-links nil

(DERIV2 STARTED "05-MAR-80 19:43:20" --DOMAIN: HEARTS --PROBLEM:
        (AVOID-TAKING-POINTS (TRICK)))


          [Initial expression:]
          (AVOID-TAKING-POINTS (TRICK))

 2:1      --- [ELABORATE by RULE124] --->
          (AVOID (TAKE-POINTS ME) (TRICK))

 2:2      --- [ELABORATE by RULE124] --->
          (ACHIEVE (NOT (DURING (TRICK) (TAKE-POINTS ME))))

                        (DURING (TRICK) (TAKE-POINTS ME))
 2:3                    --- [by RULE306] --->
                        HSM1
(HSM1 <- PROBLEM : (DURING (TRICK) (TAKE-POINTS ME)))
(HSM1 <- OBJECT : (TRICK))
(HSM1 <- CHOICE-SEQ : (CHOICE-SEQ-OF (TRICK)))

> 2:4                   --- [by RULE312] --->
                             (CONSIDER-PROP (CHOICE-SEQ-OF (TRICK)))

                                             (TRICK)
| 2:5                                        --- [ELABORATE by RULE124] --->
                                             (SCENARIO (EACH P1 (PLAYERS) (PLAY-CARD P1))
                                                       (TAKE-TRICK (TRICK-WINNER)))

                              (CHOICE-SEQ-OF
                               (SCENARIO (EACH P1 (PLAYERS) (PLAY-CARD P1))
                                          (TAKE-TRICK (TRICK-WINNER))))
| 2:6                         --- [DISTRIBUTE by RULE284] --->
                              (APPEND (CHOICE-SEQ-OF (EACH P1 (PLAYERS) (PLAY-CARD P1)))
                                      (CHOICE-SEQ-OF (TAKE-TRICK (TRICK-WINNER))))

                                      (CHOICE-SEQ-OF (TAKE-TRICK (TRICK-WINNER)))
| 2:7                                 --- [COMPUTE by RULE308] --->
                                      NIL
                              (APPEND (CHOICE-SEQ-OF (EACH P1 (PLAYERS) (PLAY-CARD P1))) NIL)
| 2:8                         --- [SIMPLIFY by RULE176] --->
                              (APPEND (CHOICE-SEQ-OF (EACH P1 (PLAYERS) (PLAY-CARD P1))))

| 2:9                         --- [SIMPLIFY by RULE178] --->
                              (CHOICE-SEQ-OF (EACH P1 (PLAYERS) (PLAY-CARD P1)))

| 2:10                        --- [DISTRIBUTE by RULE307] --->
                              (APPLY APPEND (EACH P1 (PLAYERS) (CHOICE-SEQ-OF (PLAY-CARD P1))))

                                                          (PLAY-CARD P1)
| 2:11                                                    --- [ELABORATE by RULE124] --->
                                                          (CHOOSE (CARD-OF P1)
                                                                  (LEGALCARDS P1)
                                                                  (PLAY P1 (CARD-OF P1)))

                                           (CHOICE-SEQ-OF
                                            (CHOOSE (CARD-OF P1) (LEGALCARDS P1) (PLAY P1 (CARD-OF P1))))
| 2:12                                     --- [RECOGNIZE by RULE309] --->
                                           (LIST (CHOOSE (CARD-OF P1) (LEGALCARDS P1) (PLAY P1 (CARD-OF P1))))

                              (APPLY APPEND
                                     (EACH P1 (PLAYERS)
                                           (LIST (CHOOSE (CARD-OF P1) (LEGALCARDS P1) (PLAY P1 (CARD-OF P1))))))
| 2:13                        --- [SIMPLIFY by RULE310] --->
                              (EACH P1 (PLAYERS)
                                    (CHOOSE (CARD-OF P1) (LEGALCARDS P1) (PLAY P1 (CARD-OF P1))))

               (CONSIDER-PROP
                (EACH P1 (PLAYERS)
                      (CHOOSE (CARD-OF P1) (LEGALCARDS P1) (PLAY P1 (CARD-OF P1)))))
< 2:14         --- [by RULE313] --->
          (ACHIEVE (NOT HSM1))
(HSM1 <- CHOICE-SEQ :
      (EACH P1 (PLAYERS)
            (CHOOSE (CARD-OF P1) (LEGALCARDS P1) (PLAY P1 (CARD-OF P1)))))

                        HSM1
 2:15                   --- [ELABORATE by RULE311] --->
                        HSM1
(HSM1 <- SEQUENCE : CARD-OF)
(HSM1 <- CHOICES : (PROJECT CARD-OF (PLAYERS)))
(HSM1 <- CHOICE-SETS . (LAMBDA (P1) (LEGALCARDS P1)))
(HSM1 <- INDICES : (PLAYERS))
(HSM1 <- INDEX : P1)
(HSM1 <- VARIABLES : NIL)
(HSM1 <- BINDINGS : NIL)
(HSM1 <- INITIAL-PATH : NIL)
(HSM1 <- COMPLETION-TEST . (LAMBDA (PATH) (= (# PATH) (# (PLAYERS)))))

> 2:16                  --- [by RULE314] --->
                             (CONSIDER-PROP
                              (REFORMULATE (DURING (TRICK) (TAKE-POINTS ME))
                                           (PROJECT CARD-OF (PLAYERS))))

                                            (PROJECT CARD-OF (PLAYERS))
| 2:17                                      --- [RECOGNIZE by RULE123] --->
                                            (CARDS-PLAYED)

                                            (DURING (TRICK) (TAKE-POINTS ME))
| 2:18                                      --- [REDUCE by RULE301] --->
                                            (AND [HAVE-POINTS (CARDS-PLAYED)]
                                                 [= (CARD-OF ME) (HIGHEST-IN-SUIT-LED (CARDS-PLAYED))])

                               (REFORMULATE (AND [HAVE-POINTS (CARDS-PLAYED)]
                                                 [= (CARD-OF ME) (HIGHEST-IN-SUIT-LED (CARDS-PLAYED))])
                                            (CARDS-PLAYED))
| 2:19                         --- [by RULE315] --->
                               ((LAMBDA (CARDS-PLAYED1)
                                 (AND [HAVE-POINTS CARDS-PLAYED1]
                                      [= (CARD-OF ME) (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)]))
                                (CARDS-PLAYED))

                (CONSIDER-PROP
                 ((LAMBDA (CARDS-PLAYED1)
                   (AND [HAVE-POINTS CARDS-PLAYED1]
                        [= (CARD-OF ME) (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)]))
                  (CARDS-PLAYED)))
< 2:20          --- [by RULE313] --->
           (ACHIEVE (NOT HSM1))
(HSM1 <- REFORMULATED-PROBLEM :
      ((LAMBDA (CARDS-PLAYED1)
        (AND [HAVE-POINTS CARDS-PLAYED1]
             [= (CARD-OF ME) (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)]))
       (CARDS-PLAYED)))

                          HSM1
 2:21                     --- [by RULE318] --->
                          HSM1
(HSM1 <- REFORMULATED-PROBLEM :
      ((LAMBDA (CARDS-PLAYED1)
        (AND [HAVE-POINTS CARDS-PLAYED1]
             [= (CARDS-PLAYED1 ME) (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)]))
       (CARDS-PLAYED)))

 2:22                     --- [by RULE317] --->
                          HSM1
(HSM1 <- PATH : CARDS-PLAYED1)
(HSM1 <- STEP-ORDER : NIL)
(HSM1 <- STEP-TEST : T)
(HSM1 <- PATH-ORDER : NIL)
(HSM1 <- PATH-TEST : T)
(HSM1 <- SOLUTION-TEST :
      (LAMBDA (CARDS-PLAYED1)
       (AND [HAVE-POINTS CARDS-PLAYED1]
            [= (CARDS-PLAYED1 ME) (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)])))

> 2:23                    --- [by RULE312] --->
                              (CONSIDER-PROP (LAMBDA (P1) (LEGALCARDS P1)))

                                   (LEGALCARDS P1)
| 2:24                             --- [ELABORATE by RULE124] --->
                                   (SET-OF C2 (CARDS) (LEGAL P1 C2))

                 (CONSIDER-PROP (LAMBDA (P1) (SET-OF C2 (CARDS) (LEGAL P1 C2))))
< 2:25           --- [by RULE313] --->
              (ACHIEVE (NOT HSM1))
(HSM1 <- CHOICE-SETS : (LAMBDA (P1) (SET-OF C2 (CARDS) (LEGAL P1 C2))))

                          HSM1
> 2:26                    --- [by RULE318] --->
                              (CONSIDER-PROP
                               (LAMBDA (P1) (SET-OF C2 (CARDS) (POSSIBLE (LEGAL P1 C2)))))

                                              (LEGAL P1 C2)
 | 2:27                                       --- [ELABORATE by RULE124] --->
                                              (AND [HAS P1 C2]
                                                   [=> (LEADING P1)
                                                       (OR [CAN-LEAD-HEARTS P1] [NEQ (SUIT-OF C2) H])]
                                                   [=> (FOLLOWING P1)
                                                       (OR [VOID P1 (SUIT-LED)] [IN-SUIT C2 (SUIT-LED)])])

                                                   (HAS P1 C2)
|> 2:28                                            --- [by RULE319] --->
                                        (CONSIDER-NOTE (=> (OUT C7) IF (=> (HAS P1 C2) (OUT C7))))

                                                        (OUT C7)
|| 2:29                                                 --- [ELABORATE by RULE124] --->
                                                        (EXISTS P4 (OPPONENTS ME) (HAS P4 C7))

                                                     (=> (HAS P1 C2)
                                                         (EXISTS P4 (OPPONENTS ME) (HAS P4 C7)))
||> 2:30                                             --- [by RULE322] --->
                                                (SHOW (AND [IN P4 (OPPONENTS ME)] [= P1 P4] [= C2 C7]))

                                                         (= P1 P4)
||| 2:31                                                 --- [EVAL by RULE271] --->
                                                         T
(P4 <- BINDING : P1)

                                                         (= C2 C7)
||| 2:32                                                 --- [EVAL by RULE271] --->
                                                         T
(C7 <- BINDING : C2)

                                                     (AND [IN P4 (OPPONENTS ME)] T T)
||| 2:33                                             --- [SIMPLIFY by RULE11] --->
                                                     (AND [IN P4 (OPPONENTS ME)] T)

||| 2:34                                             --- [SIMPLIFY by RULE11] --->
                                                     (AND [IN P4 (OPPONENTS ME)])
```

```
||| 2:35                                                        --- [SIMPLIFY by RULE178] --->
                                                                (IN P4 (OPPONENTS ME))

                                                                    P4
||| 2:36                                                            --- [EVAL by RULE127] --->
                                                                    P1

                                                          (SHOW (IN P1 (OPPONENTS ME)))
||< 2:37                                                  --- [GIVE-UP by RULE151] --->
                                              (CONSIDER-NOTE (=> (OUT C7) IF (IN P1 (OPPONENTS ME))))

                                                                     C7
|| 2:38                                                              --- [EVAL by RULE127] --->
                                                                     C2

                                              (CONSIDER-NOTE (=> (OUT C2) IF (IN P1 (OPPONENTS ME))))
|< 2:39                                       --- [by RULE320] --->
              (CONSIDER-PROP
               (LAMBDA (P1)
                (SET-OF C2 (CARDS)
                        (POSSIBLE (AND [HAS P1 C2]
                                       [=> (LEADING P1) (OR [CAN-LEAD-HEARTS P1] [NEQ (SUIT-OF C2) H])]
                                       [=> (FOLLOWING P1)
                                           (OR [VOID P1 (SUIT-LED)] [IN-SUIT C2 (SUIT-LED)])])))))
((HAS P1 C2) <- ; (=> (OUT C2) IF (IN P1 (OPPONENTS ME))))

                                                            (HAS P1 C2)
|> 2:40                                                     --- [by RULE319] --->
                                              (CONSIDER-NOTE
                                               (=> (NON-VOID P5) IF (=> (HAS P1 C2) (NON-VOID P5))))

                                                                    (NON-VOID P5)
|| 2:41                                                             --- [ELABORATE by RULE124] --->
                                                                    (NOT (VOID P5 (SUIT-LED)))

                                                                         (VOID P5 (SUIT-LED))
|| 2:42                                                                  --- [ELABORATE by RULE124] --->
                                                                         (NOT (EXISTS C11 (CARDS-IN-HAND P5)
                                                                                      (IN-SUIT C11 (SUIT-LED))))

                                                                    (NOT (NOT (EXISTS C11 (CARDS-IN-HAND P5)
                                                                                      (IN-SUIT C11 (SUIT-LED)))))
|| 2:43                                                             --- [SIMPLIFY by RULE88] --->
                                                                    (EXISTS C11 (CARDS-IN-HAND P5)
                                                                            (IN-SUIT C11 (SUIT-LED)))

                                                                            (CARDS-IN-HAND P5)
|| 2:44                                                                     --- [ELABORATE by RULE124] --->
                                                                            (SET-OF C12 (CARDS) (HAS P5 C12))

                                                                    (EXISTS C11 (SET-OF C12 (CARDS)
                                                                                        (HAS P5 C12))
                                                                            (IN-SUIT C11 (SUIT-LED)))
|| 2:45                                                             --- [by RULE135] --->
                                                                    (EXISTS C11 (CARDS)
                                                                            (AND [HAS P5 C11]
                                                                                 [IN-SUIT C11 (SUIT-LED)]))

                                                                (=> (HAS P1 C2)
                                                                    (EXISTS C11 (CARDS)
                                                                            (AND [HAS P5 C11]
                                                                                 [IN-SUIT C11 (SUIT-LED)])))
|| 2:46                                                         --- [by RULE220] --->
                                                                (EXISTS C11 (CARDS)
                                                                        (=> (HAS P1 C2)
                                                                            (AND [HAS P5 C11]
                                                                                 [IN-SUIT C11 (SUIT-LED)])))

                                                                        (=> (HAS P1 C2)
                                                                            (AND [HAS P5 C11]
                                                                                 [IN-SUIT C11 (SUIT-LED)]))
|| 2:47                                                                 --- [REDUCE by RULE321] --->
                                                                        (AND [=> (HAS P1 C2) (HAS P5 C11)]
                                                                             [IN-SUIT C11 (SUIT-LED)])

                                                                             (=> (HAS P1 C2) (HAS P5 C11))
||> 2:48                                                                     --- [by RULE91] --->
                                                                       (SHOW (AND [= P1 P5] [= C2 C11]))

                                                                                  (= P1 P5)
||| 2:49                                                                          --- [EVAL by RULE271] --->
```

```
                                                                                  T
(P5 <- BINDING : P1)

                                                                                  (= C2 C11)
||| 2:50                                                                          --- [EVAL by RULE271] --->
                                                                                  T
(C11 <- BINDING : C2)

                                                                             (AND T T)
||| 2:51                                                                     --- [COMPUTE by RULE236] --->
                                                                             T

                                                                       (SHOW T)
||< 2:52                                                               --- [SUCCEED by RULE53] --->
                                              (CONSIDER-NOTE
                                               (=> (NON-VOID P5)
                                                   IF (EXISTS C11 (CARDS) (AND T [IN-SUIT C11 (SUIT-LED)]))))

                                                                        (AND T [IN-SUIT C11 (SUIT-LED)])
|| 2:53                                                                 --- [SIMPLIFY by RULE11] --->
                                                                        (AND [IN-SUIT C11 (SUIT-LED)])

|| 2:54                                                                 --- [SIMPLIFY by RULE178] --->
                                                                        (IN-SUIT C11 (SUIT-LED))

                                                                                 C11
|| 2:55                                                                          --- [EVAL by RULE127] --->
                                                                                 C2

                                                                (EXISTS C11 (CARDS) (IN-SUIT C2 (SUIT-LED)))
|| 2:56                                                         --- [REMOVE-QUANT by RULE155] --->
                                                                (IN-SUIT C2 (SUIT-LED))

                                              (CONSIDER-NOTE (=> (NON-VOID P5) IF (IN-SUIT C2 (SUIT-LED))))
|< 2:57                                       --- [by RULE320] --->
               (CONSIDER-PROP
                (LAMBDA (P1)
                 (SET-OF C2 (CARDS)
                        (POSSIBLE (AND [HAS P1 C2]
                                        [=> (LEADING P1) (OR [CAN-LEAD-HEARTS P1] [NEQ (SUIT-OF C2) H])]
                                        [=> (FOLLOWING P1)
                                            (OR [VOID P1 (SUIT-LED)] [IN-SUIT C2 (SUIT-LED)])])))))
((HAS P1 C2) <- ; (=> (NON-VOID P5) IF (IN-SUIT C2 (SUIT-LED))))

< 2:58         --- [by RULE313] --->
          (ACHIEVE (NOT HSM1))
(HSM1 <- CHOICE-SETS :
      (LAMBDA (P1)
       (SET-OF C2 (CARDS)
               (POSSIBLE (AND [HAS P1 C2]
                              [=> (LEADING P1) (OR [CAN-LEAD-HEARTS P1] [NEQ (SUIT-OF C2) H])]
                              [=> (FOLLOWING P1)
                                  (OR [VOID P1 (SUIT-LED)] [IN-SUIT C2 (SUIT-LED)])])))))

                       HSM1
> 2:59                 --- [by RULE323] --->
         (CONSIDER-PROP
          (AND T
               [LAMBDA (CARDS-PLAYED1)
                (=> (=> (= ((PROJECT CARD-OF (PLAYERS)) ME)
                          (HIGHEST-IN-SUIT-LED (PROJECT CARD-OF (PLAYERS))))
                       (= (CARDS-PLAYED1 ME) (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)))
                   (= (CARDS-PLAYED1 ME) (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)))]))

                       (AND T
                            [LAMBDA (CARDS-PLAYED1)
                             (=> (=> (= ((PROJECT CARD-OF (PLAYERS)) ME)
                                        (HIGHEST-IN-SUIT-LED (PROJECT CARD-OF (PLAYERS))))
                                     (= (CARDS-PLAYED1 ME) (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)))
                                 (= (CARDS-PLAYED1 ME) (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)))])
| 2:60                 --- [SIMPLIFY by RULE11] --->
                       (AND [LAMBDA (CARDS-PLAYED1)
                             (=> (=> (= ((PROJECT CARD-OF (PLAYERS)) ME)
                                        (HIGHEST-IN-SUIT-LED (PROJECT CARD-OF (PLAYERS))))
                                     (= (CARDS-PLAYED1 ME) (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)))
                                 (= (CARDS-PLAYED1 ME) (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)))])

| 2:61                 --- [SIMPLIFY by RULE178] --->
                       (LAMBDA (CARDS-PLAYED1)
                        (=> (=> (= ((PROJECT CARD-OF (PLAYERS)) ME)
                                   (HIGHEST-IN-SUIT-LED (PROJECT CARD-OF (PLAYERS))))
                              (= (CARDS-PLAYED1 ME) (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)))
                           (= (CARDS-PLAYED1 ME) (HIGHEST-IN-SUIT-LED CARDS-PLAYED1))))

                                 (=> (= ((PROJECT CARD-OF (PLAYERS)) ME)
                                        (HIGHEST-IN-SUIT-LED (PROJECT CARD-OF (PLAYERS))))
                                     (= (CARDS-PLAYED1 ME) (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)))
|> 2:62                          --- [by RULE91] --->
                            (SHOW (AND [= ((PROJECT CARD-OF (PLAYERS)) ME) (CARDS-PLAYED1 ME)]
                                       [= (HIGHEST-IN-SUIT-LED (PROJECT CARD-OF (PLAYERS)))
                                          (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)]))

                                       (= (HIGHEST-IN-SUIT-LED (PROJECT CARD-OF (PLAYERS)))
                                          (HIGHEST-IN-SUIT-LED CARDS-PLAYED1))
||> 2:63                               --- [by RULE324] --->
                               (SHOW (SUBSET CARDS-PLAYED1 (PROJECT CARD-OF (PLAYERS))))

||< 2:64                       --- [ASSUME by RULE32] --->
                            (SHOW (AND [= ((PROJECT CARD-OF (PLAYERS)) ME) (CARDS-PLAYED1 ME)]
                                       [IN (HIGHEST-IN-SUIT-LED (PROJECT CARD-OF (PLAYERS)))
                                           CARDS-PLAYED1]))

                                         (HIGHEST-IN-SUIT-LED (PROJECT CARD-OF (PLAYERS)))
|| 2:65                                  --- [REDUCE by RULE301] --->
                                       ((PROJECT CARD-OF (PLAYERS)) ME)

                                       (= ((PROJECT CARD-OF (PLAYERS)) ME) (CARDS-PLAYED1 ME))
||> 2:66                               --- [by RULE325] --->
                               (SHOW (SUBSET CARDS-PLAYED1 (PROJECT CARD-OF (PLAYERS))))

||< 2:67                       --- [ASSUME by RULE32] --->
                            (SHOW (AND [IN ME (INDICES-OF CARDS-PLAYED1)]
                                       [IN ((PROJECT CARD-OF (PLAYERS)) ME) CARDS-PLAYED1]))

                                         ((PROJECT CARD-OF (PLAYERS)) ME)
|| 2:68                                  --- [REDUCE by RULE301] --->
                                         (CARDS-PLAYED1 ME)

                                       (IN (CARDS-PLAYED1 ME) CARDS-PLAYED1)
|| 2:69                                --- [REDUCE by RULE326] --->
                                       (IN ME (INDICES-OF CARDS-PLAYED1))

                                 (AND [IN ME (INDICES-OF CARDS-PLAYED1)]
                                      [IN ME (INDICES-OF CARDS-PLAYED1)])
|| 2:70                          --- [SIMPLIFY by RULE188] --->
                                 (AND [IN ME (INDICES-OF CARDS-PLAYED1)])

|| 2:71                          --- [SIMPLIFY by RULE178] --->
                                 (IN ME (INDICES-OF CARDS-PLAYED1))

                            (SHOW (IN ME (INDICES-OF CARDS-PLAYED1)))
|< 2:72                     --- [GIVE-UP by RULE151] --->
        (CONSIDER-PROP
         (LAMBDA (CARDS-PLAYED1)
          (=> (IN ME (INDICES-OF CARDS-PLAYED1))
              (= (CARDS-PLAYED1 ME) (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)))))

< 2:73   --- [by RULE313] --->
         (ACHIEVE (NOT HSM1))
(HSM1 <- PATH-TEST :
      (LAMBDA (CARDS-PLAYED1)
       (=> (IN ME (INDICES-OF CARDS-PLAYED1))
           (= (CARDS-PLAYED1 ME) (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)))))

                      HSM1
> 2:74                --- [by RULE327] --->
                (SHOW (= (=> (IN ME (INDICES-OF CARDS-PLAYED1))
                             (= (CARDS-PLAYED1 ME) (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)))
                         (FORALL P1 (INDICES-OF CARDS-PLAYED1) Q1)))

                                (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)
| 2:75                          --- [ELABORATE by RULE124] --->
                                (HIGHEST (CARDS-IN-SUIT-LED CARDS-PLAYED1))

| 2:76                          --- [ELABORATE by RULE124] --->
                                (THE C18 (CARDS-IN-SUIT-LED CARDS-PLAYED1)
                                    (FORALL X1 (CARDS-IN-SUIT-LED CARDS-PLAYED1)
                                           (NOT (HIGHER X1 C18))))

                              (= (CARDS-PLAYED1 ME)
                                (THE C18 (CARDS-IN-SUIT-LED CARDS-PLAYED1)
                                    (FORALL X1 (CARDS-IN-SUIT-LED CARDS-PLAYED1)
                                                 (NOT (HIGHER X1 C18)))))
| 2:77                            --- [UNBRACKET by RULE64] --->
                                 (AND [IN (CARDS-PLAYED1 ME) (CARDS-IN-SUIT-LED CARDS-PLAYED1)]
                                      [FORALL X1 (CARDS-IN-SUIT-LED CARDS-PLAYED1)
                                              (NOT (HIGHER X1 (CARDS-PLAYED1 ME)))])

                                              (CARDS-IN-SUIT-LED CARDS-PLAYED1)
| 2:78                                        --- [ELABORATE by RULE124] --->
                                              (SET-OF C19 CARDS-PLAYED1 (= (SUIT-OF C19) (SUIT-LED)))

                                      (FORALL X1 (SET-OF C19 CARDS-PLAYED1 (= (SUIT-OF C19) (SUIT-LED)))
                                              (NOT (HIGHER X1 (CARDS-PLAYED1 ME))))
| 2:79                                --- [by RULE219] --->
                                      (FORALL X1 CARDS-PLAYED1
                                              (=> (= (SUIT-OF X1) (SUIT-LED))
                                                  (NOT (HIGHER X1 (CARDS-PLAYED1 ME)))))

| 2:80                                --- [by RULE328] --->
                                      (FORALL P1 (INDICES-OF CARDS-PLAYED1)
                                              (=> (= (SUIT-OF (CARDS-PLAYED1 P1)) (SUIT-LED))
                                                  (NOT (HIGHER (CARDS-PLAYED1 P1) (CARDS-PLAYED1 ME)))))

                            (=> (IN ME (INDICES-OF CARDS-PLAYED1))
                                 (AND [IN (CARDS-PLAYED1 ME) (CARDS-IN-SUIT-LED CARDS-PLAYED1)]
                                      [FORALL P1 (INDICES-OF CARDS-PLAYED1)
                                              (=> (= (SUIT-OF (CARDS-PLAYED1 P1)) (SUIT-LED))
                                                  (NOT (HIGHER (CARDS-PLAYED1 P1) (CARDS-PLAYED1 ME))))]))
| 2:81                      --- [by RULE329] --->
                            (FORALL P1 (INDICES-OF CARDS-PLAYED1)
                                     (=> (IN ME (INDICES-OF CARDS-PLAYED1))
                                         (AND [IN (CARDS-PLAYED1 ME) (CARDS-IN-SUIT-LED CARDS-PLAYED1)]
                                              [=> (= (SUIT-OF (CARDS-PLAYED1 P1)) (SUIT-LED))
                                                  (NOT (HIGHER (CARDS-PLAYED1 P1) (CARDS-PLAYED1 ME)))])))

| 2:82                      --- [by RULE330] --->
                            (FORALL P1 (INDICES-OF CARDS-PLAYED1)
                                     (=> (NOT (AFTER ME P1))
                                         (AND [IN (CARDS-PLAYED1 ME) (CARDS-IN-SUIT-LED CARDS-PLAYED1)]
                                              [=> (= (SUIT-OF (CARDS-PLAYED1 P1)) (SUIT-LED))
                                                  (NOT (HIGHER (CARDS-PLAYED1 P1) (CARDS-PLAYED1 ME)))])))

                          (= (FORALL P1 (INDICES-OF CARDS-PLAYED1)
                                     (=> (NOT (AFTER ME P1))
                                         (AND [IN (CARDS-PLAYED1 ME) (CARDS-IN-SUIT-LED CARDS-PLAYED1)]
                                              [=> (= (SUIT-OF (CARDS-PLAYED1 P1)) (SUIT-LED))
                                                  (NOT (HIGHER (CARDS-PLAYED1 P1) (CARDS-PLAYED1 ME)))])))
                            (FORALL P1 (INDICES-OF CARDS-PLAYED1) Q1))
| 2:83                    --- [REDUCE by RULE192] --->
                          (= (=> (NOT (AFTER ME P1))
                                 (AND [IN (CARDS-PLAYED1 ME) (CARDS-IN-SUIT-LED CARDS-PLAYED1)]
                                      [=> (= (SUIT-OF (CARDS-PLAYED1 P1)) (SUIT-LED))
                                          (NOT (HIGHER (CARDS-PLAYED1 P1) (CARDS-PLAYED1 ME)))]))
                            Q1)

| 2:84                    --- [EVAL by RULE271] --->
                          T
(Q1 <- BINDING :
     (=> (NOT (AFTER ME P1))
         (AND [IN (CARDS-PLAYED1 ME) (CARDS-IN-SUIT-LED CARDS-PLAYED1)]
              [=> (= (SUIT-OF (CARDS-PLAYED1 P1)) (SUIT-LED))
                  (NOT (HIGHER (CARDS-PLAYED1 P1) (CARDS-PLAYED1 ME)))])))

                   (SHOW T)
< 2:85             --- [SUCCEED by RULE34] --->
            (ACHIEVE (NOT (THEN $CONSIDER-PROP HSM1 STEP-TEST
                                (AND T [LAMBDA (P1 C21) (THEN $SUBST C21 (CARDS-PLAYED1 P1) Q1)]))))

                                (AND T [LAMBDA (P1 C21) (THEN $SUBST C21 (CARDS-PLAYED1 P1) Q1)])
  2:86                          --- [SIMPLIFY by RULE11] --->
                                (AND [LAMBDA (P1 C21) (THEN $SUBST C21 (CARDS-PLAYED1 P1) Q1)])

  2:87                          --- [SIMPLIFY by RULE178] --->
                                (LAMBDA (P1 C21) (THEN $SUBST C21 (CARDS-PLAYED1 P1) Q1))

                                             Q1
  2:88                                       --- [EVAL by RULE127] --->
                                             (=> (NOT (AFTER ME P1))
                                                 (AND [IN (CARDS-PLAYED1 ME) (CARDS-IN-SUIT-LED CARDS-PLAYED1)]
                                                      [=> (= (SUIT-OF (CARDS-PLAYED1 P1)) (SUIT-LED))
                                                          (NOT (HIGHER (CARDS-PLAYED1 P1) (CARDS-PLAYED1 ME)))]))
```

```
                                    (THEN $SUBST C21 (CARDS-PLAYED1 P1)
                                          (=> (NOT (AFTER ME P1))
                                              (AND [IN (CARDS-PLAYED1 ME) (CARDS-IN-SUIT-LED CARDS-PLAYED1)]
                                                   [=> (= (SUIT-OF (CARDS-PLAYED1 P1)) (SUIT-LED))
                                                       (NOT (HIGHER (CARDS-PLAYED1 P1) (CARDS-PLAYED1 ME)))]))))
 2:89                               --- [SIMPLIFY by RULE331] --->
                                    (=> (NOT (AFTER ME P1))
                                        (AND [IN (CARDS-PLAYED1 ME) (CARDS-IN-SUIT-LED CARDS-PLAYED1)]
                                             [=> (= (SUIT-OF C21) (SUIT-LED))
                                                 (NOT (HIGHER C21 (CARDS-PLAYED1 ME)))]))

                                                 (CARDS-IN-SUIT-LED CARDS-PLAYED1)
 2:90                                            --- [ELABORATE by RULE124] --->
                                                 (SET-OF C22 CARDS-PLAYED1 (= (SUIT-OF C22) (SUIT-LED)))

                                             (IN (CARDS-PLAYED1 ME)
                                                 (SET-OF C22 CARDS-PLAYED1 (= (SUIT-OF C22) (SUIT-LED))))
 2:91                                        --- [REMOVE-QUANT by RULE10] --->
                                             (AND [IN (CARDS-PLAYED1 ME) CARDS-PLAYED1]
                                                  [= (SUIT-OF (CARDS-PLAYED1 ME)) (SUIT-LED)])

 2:92                                        --- [REDUCE by RULE301] --->
                                             (= (SUIT-OF (CARDS-PLAYED1 ME)) (SUIT-LED))

                       (THEN $CONSIDER-PROP HSM1 STEP-TEST
                             (LAMBDA (P1 C21)
                              (=> (NOT (AFTER ME P1))
                                  (AND [= (SUIT-OF (CARDS-PLAYED1 ME)) (SUIT-LED)]
                                       [=> (= (SUIT-OF C21) (SUIT-LED))
                                           (NOT (HIGHER C21 (CARDS-PLAYED1 ME)))]))))
 2:93                  --- [by RULE332] --->
                       (THEN $CONSIDER-PROP HSM1 STEP-TEST
                             (LAMBDA (P1 C21)
                              (=> (NOT (AFTER ME P1))
                                  (AND [= (SUIT-OF MY-CARD4) (SUIT-LED)]
                                       [=> (= (SUIT-OF C21) (SUIT-LED)) (NOT (HIGHER C21 MY-CARD4))]))))
(HSM1 <- VARIABLES : (MY-CARD4))
(HSM1 <- BINDINGS : ((CARD-OF ME)))
(HSM1 <- INITIAL-PATH : (PROJECT CARD-OF (PREFIX (PLAYERS) ME)))

 2:94                  --- [by RULE333] --->
                       (THEN $CONSIDER-PROP HSM1 STEP-TEST
                             (LAMBDA (P1 C21)
                              (=> (NOT (AFTER ME P1))
                                  (AND [= (SUIT-OF MY-CARD4) SUIT-LED2]
                                       [=> (= (SUIT-OF C21) SUIT-LED2) (NOT (HIGHER C21 MY-CARD4))]))))
(HSM1 <- BINDINGS : ((SUIT-LED) (CARD-OF ME)))
(HSM1 <- VARIABLES : (SUIT-LED2 MY-CARD4))

 2:95                  --- [SIMPLIFY by RULE334] --->
                       HSM1
(HSM1 <- STEP-TEST :
      (LAMBDA (P1 C21)
       (=> (NOT (AFTER ME P1))
           (AND [= (SUIT-OF MY-CARD4) SUIT-LED2]
                [=> (= (SUIT-OF C21) SUIT-LED2) (NOT (HIGHER C21 MY-CARD4))]))))

> 2:96                 --- [by RULE335] --->
                 (SHOW (=> (HAVE-POINTS (PREFIX CARDS-PLAYED1 ANY))
                           (HAVE-POINTS CARDS-PLAYED1)))

                       (=> (HAVE-POINTS (PREFIX CARDS-PLAYED1 ANY))
                           (HAVE-POINTS CARDS-PLAYED1))
|> 2:97                --- [REDUCE by RULE336] --->
                 (SHOW (SUBSET (PREFIX CARDS-PLAYED1 ANY) CARDS-PLAYED1))

                       (SUBSET (PREFIX CARDS-PLAYED1 ANY) CARDS-PLAYED1)
|| 2:98                --- [EVAL by RULE337] --->
                       T

                 (SHOW T)
|< 2:99          --- [SUCCEED by RULE63] --->
                 (SHOW T)

< 2:100          --- [SUCCEED by RULE34] --->
          (ACHIEVE (NOT HSM1))

                       HSM1
> 2:101                --- [by RULE338] --->
         (CONSIDER-PROP
          (LAMBDA (P1 C21) (HAVE-POINTS (APPEND CARDS-PLAYED1 (LIST C21)))))
```

```
                                (HAVE-POINTS (APPEND CARDS-PLAYED1 (LIST C21)))
| 2:102                         --- [DISTRIBUTE by RULE284] --->
                                (OR [HAVE-POINTS CARDS-PLAYED1] [HAVE-POINTS (LIST C21)])

                                    (HAVE-POINTS (LIST C21))
| 2:103                             --- [ELABORATE by RULE124] --->
                                    (EXISTS C23 (LIST C21) (HAS-POINTS C23))

| 2:104                             --- [SIMPLIFY by RULE266] --->
                                    (HAS-POINTS C21)

         (CONSIDER-PROP
          (LAMBDA (P1 C21) (OR [HAVE-POINTS CARDS-PLAYED1] [HAS-POINTS C21])))
< 2:105  --- [by RULE313] --->
          (ACHIEVE (NOT HSM1))
(HSM1 <- STEP-ORDER :
      (LAMBDA (P1 C21) (OR [HAVE-POINTS CARDS-PLAYED1] [HAS-POINTS C21])))

          [Final expression:]
          (ACHIEVE (NOT HSM1))
NIL
<72>p hsm1

(HSM1 WITH (PROBLEM : (DURING (TRICK) (TAKE-POINTS ME)))
      (OBJECT : (TRICK))
      (CHOICE-SEQ :
                  (EACH P1 (PLAYERS)
                        (CHOOSE (CARD-OF P1)
                                (LEGALCARDS P1)
                                (PLAY P1 (CARD-OF P1)))))
      (SEQUENCE : CARD-OF)
      (CHOICES : (PROJECT CARD-OF (PLAYERS)))
      (CHOICE-SETS :
                   (LAMBDA (P1)
                    (SET-OF C2 (CARDS)
                            (POSSIBLE (AND [HAS P1 C2]
                                            [=> (LEADING P1)
                                                (OR [CAN-LEAD-HEARTS P1]
                                                    [NEQ (SUIT-OF C2) H])]
                                            [=> (FOLLOWING P1)
                                                (OR [VOID P1 (SUIT-LED)]
                                                    [IN-SUIT C2 (SUIT-LED)])]))
                     )))
      (INDICES : (PLAYERS))
      (INDEX : P1)
      (COMPLETION-TEST : (LAMBDA (PATH) (= (# PATH) (# (PLAYERS)))))
      (REFORMULATED-PROBLEM :
       ((LAMBDA (CARDS-PLAYED1)
         (AND [HAVE-POINTS CARDS-PLAYED1]
              [= (CARDS-PLAYED1 ME) (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)]))
        (CARDS-PLAYED)))
      (PATH : CARDS-PLAYED1)
      (STEP-ORDER :
                  (LAMBDA (P1 C21)
                   (OR [HAVE-POINTS CARDS-PLAYED1] [HAS-POINTS C21])))
      (STEP-TEST :
                 (LAMBDA (P1 C21)
                  (=> (NOT (AFTER ME P1))
                      (AND [= (SUIT-OF MY-CARD4) SUIT-LED2]
                           [=> (= (SUIT-OF C21) SUIT-LED2)
                               (NOT (HIGHER C21 MY-CARD4))]))))
      (PATH-ORDER : (LAMBDA (CARDS-PLAYED1) (HAVE-POINTS CARDS-PLAYED1)))
      (PATH-TEST :
                 (LAMBDA (CARDS-PLAYED1)
                  (=> (IN ME (INDICES-OF CARDS-PLAYED1))
                      (= (CARDS-PLAYED1 ME)
                         (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)))))
      (SOLUTION-TEST :
       (LAMBDA (CARDS-PLAYED1)
        (AND [HAVE-POINTS CARDS-PLAYED1]
             [= (CARDS-PLAYED1 ME) (HIGHEST-IN-SUIT-LED CARDS-PLAYED1)])))
      (VARIABLES : (SUIT-LED2 MY-CARD4))
      (BINDINGS : ((SUIT-LED) (CARD-OF ME)))
      (INITIAL-PATH : (PROJECT CARD-OF (PREFIX (PLAYERS) ME))))
HSM1
<73>p node184

(NODE184 (HAS P1 C2)
           ; (=> (NON-VOID P5) IF (IN-SUIT C2 (SUIT-LED)))
         (=> (OUT C2) IF (IN P1 (OPPONENTS ME))))
```

```
NODE184
<74>recordfile

RECORD FILE DSK: (DV2D07 . PZG) CLOSED 07-DEC-80 18:11:01
```

D.3. DERIV3: Flush the Queen

```
RECORD FILE DSK: (DV3D07 . PZG) OPENED 20-MAR-81 21:51:19
NIL
<50>p-links nil

(DERIV3 STARTED "07-APR-80 22:32:36" --DOMAIN: HEARTS --PROBLEM:
        (ACHIEVE (FLUSH QS)))


          [Initial expression:]
          (ACHIEVE (FLUSH QS))

                   (FLUSH QS)
 3:1               --- [ELABORATE by RULE124] --->
                   (MUST (OWNER-OF QS) (PLAY (OWNER-OF QS) QS))

                                (OWNER-OF QS)
 3:2                            --- [RECOGNIZE by RULE123] --->
                                (QSO)

                          (OWNER-OF QS)
 3:3                      --- [RECOGNIZE by RULE123] --->
                          (QSO)

                   (MUST (QSO) (PLAY (QSO) QS))
 3:4               --- [ELABORATE by RULE124] --->
                   (= (ACTIONS-OF (QSO)) (PLAY (QSO) QS))

                      (ACTIONS-OF (QSO))
 3:5                  --- [REDUCE by RULE61] --->
                      (PLAY-CARD (QSO))

 3:6                  --- [ELABORATE by RULE124] --->
                      (CHOOSE (CARD-OF (QSO))
                              (LEGALCARDS (QSO))
                              (PLAY (QSO) (CARD-OF (QSO))))

                   (= (CHOOSE (CARD-OF (QSO))
                              (LEGALCARDS (QSO))
                              (PLAY (QSO) (CARD-OF (QSO))))
                      (PLAY (QSO) QS))
 3:7               --- [REDUCE by RULE339] --->
                   (= (LEGALCARDS (QSO)) (SET QS))

 3:12              --- [ELABORATE by RULE340] --->
                   (AND [SUBSET (LEGALCARDS (QSO)) (SET QS)]
                        [SUBSET (SET QS) (LEGALCARDS (QSO))])

                        (SUBSET (SET QS) (LEGALCARDS (QSO)))
> 3:13                  --- [by RULE268] --->
              (CONSIDER (SUBSET (SET QS) (LEGALCARDS (QSO))))

                        (SUBSET (SET QS) (LEGALCARDS (QSO)))
| 3:14                  --- [DISTRIBUTE by RULE17] --->
                        (AND [IN QS (LEGALCARDS (QSO))])

| 3:15                  --- [SIMPLIFY by RULE178] --->
                        (IN QS (LEGALCARDS (QSO)))

                            (LEGALCARDS (QSO))
| 3:16                      --- [ELABORATE by RULE124] --->
                            (SET-OF C1 (CARDS) (LEGAL (QSO) C1))

                        (IN QS (SET-OF C1 (CARDS) (LEGAL (QSO) C1)))
| 3:17                  --- [REMOVE-QUANT by RULE10] --->
                        (AND [IN QS (CARDS)] [LEGAL (QSO) QS])

                             (IN QS (CARDS))
```

```
| 3:18                        --- [EVAL by RULE62] --->
                              T

                        (AND T [LEGAL (QSO) QS])
| 3:19                  --- [SIMPLIFY by RULE341] --->
                        (LEGAL (QSO) QS)

| 3:20                  --- [ELABORATE by RULE124] --->
                        (AND [HAS (QSO) QS]
                             [=> (LEADING (QSO))
                                 (OR [CAN-LEAD-HEARTS (QSO)] [NEQ (SUIT-OF QS) H])]
                             [=> (FOLLOWING (QSO))
                                 (OR [VOID (QSO) (SUIT-LED)] [IN-SUIT QS (SUIT-LED)])])

                                  (QSO)
| 3:21                            --- [ELABORATE by RULE124] --->
                                  (OWNER-OF QS)

                             (HAS (OWNER-OF QS) QS)
| 3:22                       --- [EVAL by RULE1] --->
                             T

                        (AND T
                             [=> (LEADING (QSO))
                                 (OR [CAN-LEAD-HEARTS (QSO)] [NEQ (SUIT-OF QS) H])]
                             [=> (FOLLOWING (QSO))
                                 (OR [VOID (QSO) (SUIT-LED)] [IN-SUIT QS (SUIT-LED)])])
| 3:23                  --- [SIMPLIFY by RULE11] --->
                        (AND [=> (LEADING (QSO))
                                 (OR [CAN-LEAD-HEARTS (QSO)] [NEQ (SUIT-OF QS) H])]
                             [=> (FOLLOWING (QSO))
                                 (OR [VOID (QSO) (SUIT-LED)] [IN-SUIT QS (SUIT-LED)])])

                                     (IN-SUIT QS (SUIT-LED))
| 3:24                               --- [ELABORATE by RULE124] --->
                                     (= (SUIT-OF QS) (SUIT-LED))

                                        (SUIT-OF QS)
| 3:25                                  --- [EVAL by RULE2] --->
                                        S

                                     (= S (SUIT-LED))
| 3:26                               --- [by RULE342] --->
                                     T
((SUIT-LED) <- ; (-> S IF (= (SUIT-LED) S)))

                                 (OR [VOID (QSO) (SUIT-LED)] T)
| 3:27                           --- [EVAL by RULE184] --->
                                 T

                             (=> (FOLLOWING (QSO)) T)
| 3:28                       --- [EVAL by RULE47] --->
                             T

                        (AND [=> (LEADING (QSO))
                                 (OR [CAN-LEAD-HEARTS (QSO)] [NEQ (SUIT-OF QS) H])]
                             T)
| 3:29                  --- [SIMPLIFY by RULE11] --->
                        (AND [=> (LEADING (QSO))
                                 (OR [CAN-LEAD-HEARTS (QSO)] [NEQ (SUIT-OF QS) H])])

                                          (SUIT-OF QS)
| 3:30                                    --- [EVAL by RULE2] --->
                                          S

                                     (NEQ S H)
| 3:31                               --- [COMPUTE by RULE236] --->
                                     T

                                 (OR [CAN-LEAD-HEARTS (QSO)] T)
| 3:32                           --- [EVAL by RULE184] --->
                                 T

                             (=> (LEADING (QSO)) T)
| 3:33                       --- [EVAL by RULE47] --->
                             T

                        (AND T)
| 3:34                  --- [COMPUTE by RULE236] --->
                        T
```

```
              (CONSIDER T)
< 3:35        --- [by RULE269] --->
          (ACHIEVE (AND [SUBSET (LEGALCARDS (QSO)) (SET QS)] T))

                   (AND [SUBSET (LEGALCARDS (QSO)) (SET QS)] T)
 3:36              --- [SIMPLIFY by RULE343] --->
                   (SUBSET (LEGALCARDS (QSO)) (SET QS))

 3:37              --- [TRY-THIS by RULE4] --->
                   (EMPTY (DIFF (LEGALCARDS (QSO)) (SET QS)))

          (ACHIEVE (EMPTY (DIFF (LEGALCARDS (QSO)) (SET QS))))
 3:38     --- [REDUCE by RULE8] --->
          (UNTIL (PLAYED! QS)
                 (ACHIEVE (REMOVE-1-FROM (DIFF (LEGALCARDS (QSO)) (SET QS)))))

                                               (LEGALCARDS (QSO))
 3:39                                          --- [ELABORATE by RULE124] --->
                                               (SET-OF C3 (CARDS) (LEGAL (QSO) C3))

                                         (DIFF (SET-OF C3 (CARDS) (LEGAL (QSO) C3)) (SET QS))
 3:40                                    --- [COLLECT by RULE5] --->
                                         (SET-OF C3 (DIFF (CARDS) (SET QS)) (LEGAL (QSO) C3))

                          (REMOVE-1-FROM
                           (SET-OF C3 (DIFF (CARDS) (SET QS)) (LEGAL (QSO) C3)))
 3:41                     --- [REDUCE by RULE7] --->
                          (UNDO (AND [IN C3 (DIFF (CARDS) (SET QS))] [LEGAL (QSO) C3]))

 3:42                     --- [REDUCE by RULE36] --->
                          (AND [UNDO (LEGAL (QSO) C3)] [IN C3 (DIFF (CARDS) (SET QS))])

                                     (LEGAL (QSO) C3)
 3:43                                --- [ELABORATE by RULE124] --->
                                     (AND [HAS (QSO) C3]
                                          [=> (LEADING (QSO))
                                              (OR [CAN-LEAD-HEARTS (QSO)] [NEQ (SUIT-OF C3) H])]
                                          [=> (FOLLOWING (QSO))
                                              (OR [VOID (QSO) (SUIT-LED)] [IN-SUIT C3 (SUIT-LED)])])

                               (UNDO (AND [HAS (QSO) C3]
                                          [=> (LEADING (QSO))
                                              (OR [CAN-LEAD-HEARTS (QSO)] [NEQ (SUIT-OF C3) H])]
                                          [=> (FOLLOWING (QSO))
                                              (OR [VOID (QSO) (SUIT-LED)] [IN-SUIT C3 (SUIT-LED)])]))
 3:44                          --- [REDUCE by RULE35] --->
                               (AND [UNDO (HAS (QSO) C3)]
                                    [=> (LEADING (QSO))
                                        (OR [CAN-LEAD-HEARTS (QSO)] [NEQ (SUIT-OF C3) H])]
                                    [=> (FOLLOWING (QSO))
                                        (OR [VOID (QSO) (SUIT-LED)] [IN-SUIT C3 (SUIT-LED)])])

                          (AND [AND [UNDO (HAS (QSO) C3)]
                                    [=> (LEADING (QSO))
                                        (OR [CAN-LEAD-HEARTS (QSO)] [NEQ (SUIT-OF C3) H])]
                                    [=> (FOLLOWING (QSO))
                                        (OR [VOID (QSO) (SUIT-LED)] [IN-SUIT C3 (SUIT-LED)])]]
                               [IN C3 (DIFF (CARDS) (SET QS))])
 3:45                     --- [SIMPLIFY by RULE177] --->
                          (AND [UNDO (HAS (QSO) C3)]
                               [=> (LEADING (QSO))
                                   (OR [CAN-LEAD-HEARTS (QSO)] [NEQ (SUIT-OF C3) H])]
                               [=> (FOLLOWING (QSO))
                                   (OR [VOID (QSO) (SUIT-LED)] [IN-SUIT C3 (SUIT-LED)])]
                               [IN C3 (DIFF (CARDS) (SET QS))])

                               (UNDO (HAS (QSO) C3))
> 3:46                         --- [by RULE254] --->
                         (SHOW (=> (PLAY P1 C4) (UNDO (HAS (QSO) C3))))

                                          (HAS (QSO) C3)
| 3:47                                    --- [ELABORATE by RULE124] --->
                                          (AT C3 (HAND (QSO)))

| 3:48                                    --- [ELABORATE by RULE124] --->
                                          (= (LOC C3) (HAND (QSO)))

                                    (UNDO (= (LOC C3) (HAND (QSO))))
| 3:49                              --- [by RULE344] --->
                                    (AND [= (LOC C3) (HAND (QSO))] [BECOME (LOC C3) LOC3])
```

```
| 3:50                            --- [RECOGNIZE by RULE123] --->
                                 (MOVE C3 (HAND (QSO)) LOC3)

                                 (PLAY P1 C4)
| 3:51                            --- [ELABORATE by RULE124] --->
                                 (MOVE C4 (HAND P1) POT)

                             (=> (MOVE C4 (HAND P1) POT) (MOVE C3 (HAND (QSO)) LOC3))
| 3:52                        --- [DISTRIBUTE by RULE43] --->
                             (AND [= C4 C3] [= (HAND P1) (HAND (QSO))] [= POT LOC3])

                                  (= C4 C3)
| 3:53                             --- [REDUCE by RULE194] --->
                                  T
(C4 <- BINDING : C3)

                                  (= POT LOC3)
| 3:54                             --- [EVAL by RULE271] --->
                                  T
(LOC3 <- BINDING : POT)

                                  (= (HAND P1) (HAND (QSO)))
| 3:55                             --- [DISTRIBUTE by RULE43] --->
                                  (AND [= P1 (QSO)])

                                       (= P1 (QSO))
| 3:56                                  --- [REDUCE by RULE194] --->
                                       T
(P1 <- BINDING : (QSO))

                                  (AND T)
| 3:57                             --- [COMPUTE by RULE236] --->
                                  T

                             (AND T T T)
| 3:58                        --- [COMPUTE by RULE236] --->
                             T

                        (SHOW T)
< 3:59                   --- [SUCCEED by RULE34] --->
              (UNTIL (PLAYED! QS)
                     (ACHIEVE (AND [PLAY P1 C4]
                                   [=> (LEADING (QSO))
                                       (OR [CAN-LEAD-HEARTS (QSO)] [NEQ (SUIT-OF C3) H])]
                                   [=> (FOLLOWING (QSO))
                                       (OR [VOID (QSO) (SUIT-LED)] [IN-SUIT C3 (SUIT-LED)])]
                                   [IN C3 (DIFF (CARDS) (SET QS))])))

                                   P1
 3:60                               --- [EVAL by RULE127] --->
                                   (QSO)

                                   C4
 3:61                               --- [EVAL by RULE127] --->
                                   C3

                                          (SUIT-LED)
 3:62                                      --- [EVAL by RULE346] --->
                                          S

                                             (SUIT-LED)
 3:63                                         --- [EVAL by RULE346] --->
                                          S

                                    (VOID (QSO) S)
> 3:64                               --- [by RULE268] --->
                           (CONSIDER (VOID (QSO) S))

                                    (VOID (QSO) S)
| 3:65                               --- [ELABORATE by RULE124] --->
                                    (NOT (EXISTS C5 (CARDS-IN-HAND (QSO)) (IN-SUIT C5 S)))

                                                (CARDS-IN-HAND (QSO))
| 3:66                                           --- [ELABORATE by RULE124] --->
                                                (SET-OF C6 (CARDS) (HAS (QSO) C6))

                                        (EXISTS C6 (SET-OF C6 (CARDS) (HAS (QSO) C6)) (IN-SUIT C5 S))
| 3:67                                   --- [by RULE135] --->
                                        (EXISTS C5 (CARDS) (AND [HAS (QSO) C5] [IN-SUIT C5 S]))

                                                       (QSO)
```

```
| 3:68                                                       --- [ELABORATE by RULE124] --->
                                                             (OWNER-OF QS)

                                            (EXISTS C5 (CARDS) (AND [HAS (OWNER-OF QS) C5] [IN-SUIT C5 S]))
|> 3:69                                     --- [by RULE347] --->
                                     (SHOW (AND [HAS (OWNER-OF QS) QS] [IN-SUIT QS S]))

                                                 (HAS (OWNER-OF QS) QS)
|| 3:70                                          --- [EVAL by RULE1] --->
                                                 T

                                                 (IN-SUIT QS S)
|| 3:71                                          --- [ELABORATE by RULE124] --->
                                                 (= (SUIT-OF QS) S)

                                                    (SUIT-OF QS)
|| 3:72                                             --- [EVAL by RULE2] --->
                                                    S

                                                 (= S S)
|| 3:73                                          --- [EVAL by RULE179] --->
                                                 T

                                            (AND T T)
|| 3:74                                     --- [COMPUTE by RULE236] --->
                                            T

                                      (SHOW T)
|< 3:75                               --- [SUCCEED by RULE63] --->
                              (CONSIDER (NOT T))

                                       (NOT T)
| 3:76                                 --- [COMPUTE by RULE236] --->
                                       NIL

                              (CONSIDER NIL)
< 3:77                        --- [by RULE269] --->
          (UNTIL (PLAYED! QS)
                (ACHIEVE (AND [PLAY (QSO) C3]
                              [=> (LEADING (QSO))
                                  (OR [CAN-LEAD-HEARTS (QSO)] [NEQ (SUIT-OF C3) H])]
                              [=> (FOLLOWING (QSO)) (OR NIL [IN-SUIT C3 S])]
                              [IN C3 (DIFF (CARDS) (SET QS))])))

                                  (OR NIL [IN-SUIT C3 S])
 3:78                             --- [SIMPLIFY by RULE176] --->
                                  (OR [IN-SUIT C3 S])

 3:79                             --- [SIMPLIFY by RULE178] --->
                                  (IN-SUIT C3 S)

                              (=> (LEADING (QSO))
                                  (OR [CAN-LEAD-HEARTS (QSO)] [NEQ (SUIT-OF C3) H]))
 3:80                         --- [ASSUME by RULE349] --->
                              T
((LEADING (QSO)) <- : (-> NIL IF (= (LEADING (QSO)) NIL)))

                (ACHIEVE (AND [PLAY (QSO) C3]
                              T [=> (FOLLOWING (QSO)) (IN-SUIT C3 S)]
                              [IN C3 (DIFF (CARDS) (SET QS))]))
 3:81           --- [by RULE350] --->
                (ACHIEVE (AND [PLAY (QSO) C3] T [=> (FOLLOWING (QSO)) (IN-SUIT C3 S)]))

                         (AND [PLAY (QSO) C3] T [=> (FOLLOWING (QSO)) (IN-SUIT C3 S)])
 3:82                    --- [SIMPLIFY by RULE11] --->
                         (AND [PLAY (QSO) C3] [=> (FOLLOWING (QSO)) (IN-SUIT C3 S)])

                                  (FOLLOWING (QSO))
 3:83                             --- [ELABORATE by RULE124] --->
                                  (NOT (LEADING (QSO)))

                                       (LEADING (QSO))
 3:84                                  --- [EVAL by RULE346] --->
                                       NIL

                                  (NOT NIL)
 3:85                             --- [COMPUTE by RULE236] --->
                                  T

                              (=> T (IN-SUIT C3 S))
 3:86                         --- [SIMPLIFY by RULE12] --->
```

```
                             (IN-SUIT C3 S)

                        (AND [PLAY (QSO) C3] [IN-SUIT C3 S])
 3:87                   --- [RECOGNIZE by RULE123] --->
                        (PLAY-SPADE (QSO))

                  (ACHIEVE (PLAY-SPADE (QSO)))
 3:88             --- [by RULE351] --->
                  (ACHIEVE (AND [= (LEADING (QSO)) NIL] [= (SUIT-LED) S]))

                             (= (LEADING (QSO)) NIL)
 3:89                        --- [SIMPLIFY by RULE352] --->
                             (NOT (LEADING (QSO)))

                                  (LEADING (QSO))
 3:90                             --- [ELABORATE by RULE124] --->
                                  (= (LEADER) (QSO))

                             (NOT (= (LEADER) (QSO)))
 3:91                        --- [by RULE353] --->
                             (= (LEADER) ME)

                                (SUIT-LED)
 3:92                           --- [ELABORATE by RULE124] --->
                                (SUIT-OF (CARD-OF (LEADER)))

                             (= (SUIT-OF (CARD-OF (LEADER))) S)
 3:93                        --- [RECOGNIZE by RULE123] --->
                             (IN-SUIT (CARD-OF (LEADER)) S)

 3:94                        --- [RECOGNIZE by RULE123] --->
                             (SPADE! (CARD-OF (LEADER)))

                        (AND [= (LEADER) ME] [SPADE! (CARD-OF (LEADER))])
 3:95                   --- [by RULE354] --->
                        (AND [= (LEADER) ME] [SPADE! (CARD-OF ME)])

                             (= (LEADER) ME)
 3:96                        --- [RECOGNIZE by RULE123] --->
                             (LEADING ME)

                        (AND [LEADING ME] [SPADE! (CARD-OF ME)])
 3:97                   --- [RECOGNIZE by RULE123] --->
                        (LEAD-SPADE ME)

         [Final expression:]
         (UNTIL (PLAYED! QS) (ACHIEVE (LEAD-SPADE ME)))

(ASSUMING (LEADING (QSO)) -> NIL)

(ASSUMING (SUIT-LED) -> S)
NIL
<51>recordfile

RECORD FILE DSK: (DV3D07 . PZG) CLOSED 20-MAR-81 21:56:38
```

D.4. DERIV4: Decide if the Queen is Out

```
RECORD FILE DSK: (DV4D07 . PZG) OPENED 07-DEC-80 18:20:69
NIL
<42>p-links nil

(DERIV4 STARTED "11-APR-80 23:14:14" --DOMAIN: HEARTS --PROBLEM:
        (EVAL (QS-OUT)))


         [Initial expression:]
         (EVAL (QS-OUT))

               (QS-OUT)
 4:1           --- [ELABORATE by RULE124] --->
               (OUT QS)

 4:2           --- [ELABORATE by RULE124] --->
```

```
                (EXISTS P1 (OPPONENTS ME) (HAS P1 QS))

                          (HAS P1 QS)
 4:3                      --- [ELABORATE by RULE124] --->
                          (AT QS (HAND P1))

 4:4                      --- [ELABORATE by RULE124] --->
                          (= (LOC QS) (HAND P1))

                (EXISTS P1 (OPPONENTS ME) (= (LOC QS) (HAND P1)))
 4:5            --- [COLLECT by RULE161] --->
                (EXISTS Y1 (PROJECT HAND (OPPONENTS ME)) (= (LOC QS) Y1))

 4:6            --- [REMOVE-QUANT by RULE162] --->
                (IN (LOC QS) (PROJECT HAND (OPPONENTS ME)))

 4:7            --- [TRY-THIS by RULE169] --->
                (NOT (IN (LOC QS) (DIFF (RANGE LOC) (PROJECT HAND (OPPONENTS ME)))))

                           (DIFF (RANGE LOC) (PROJECT HAND (OPPONENTS ME)))
> 4:8                      --- [by RULE268] --->
               (CONSIDER (DIFF (RANGE LOC) (PROJECT HAND (OPPONENTS ME))))

                                  (RANGE LOC)
| 4:9                             --- [ENUMERATE by RULE163] --->
                                  (PARTITION (HANDS (PLAYERS))
                                             (PILES (PLAYERS))
                                             (SET DECK POT HOLE))

| 4:10                            --- [GENERALIZE by RULE15] --->
                                  (UNION (HANDS (PLAYERS)) (PILES (PLAYERS)) (SET DECK POT HOLE))
((DISJOINT (HANDS (PLAYERS))
           (PILES (PLAYERS))
           (SET DECK POT HOLE))
 <- ;
 (-> T IF
     (= (DISJOINT (HANDS (PLAYERS))
                  (PILES (PLAYERS))
                  (SET DECK POT HOLE))
        T)))

                                             (OPPONENTS ME)
| 4:11                                       --- [ELABORATE by RULE124] --->
                                             (DIFF (PLAYERS) (SET ME))

                                  (PROJECT HAND (DIFF (PLAYERS) (SET ME)))
| 4:12                            --- [DISTRIBUTE by RULE170] --->
                                  (DIFF (PROJECT HAND (PLAYERS)) (PROJECT HAND (SET ME)))

                                        (PROJECT HAND (SET ME))
| 4:13                                  --- [DISTRIBUTE by RULE171] --->
                                        (SET (HAND ME))

                                        (PROJECT HAND (PLAYERS))
| 4:14                                  --- [RECOGNIZE by RULE123] --->
                                        (HANDS (PLAYERS))

                           (DIFF (UNION (HANDS (PLAYERS)) (PILES (PLAYERS)) (SET DECK POT HOLE))
                                 (DIFF (HANDS (PLAYERS)) (SET (HAND ME))))
| 4:15                     --- [DISTRIBUTE by RULE164] --->
                           (UNION (DIFF (UNION (HANDS (PLAYERS)) (PILES (PLAYERS)) (SET DECK POT HOLE))
                                        (HANDS (PLAYERS)))
                                  (INTERSECT (UNION (HANDS (PLAYERS)) (PILES (PLAYERS)) (SET DECK POT HOLE))
                                             (SET (HAND ME))))

                                  (DIFF (UNION (HANDS (PLAYERS)) (PILES (PLAYERS)) (SET DECK POT HOLE))
                                        (HANDS (PLAYERS)))
|> 4:16                           --- [SIMPLIFY by RULE19] --->
                            (SHOW (DISJOINT (HANDS (PLAYERS))
                                            (PILES (PLAYERS))
                                            (SET DECK POT HOLE)))

                                  (DISJOINT (HANDS (PLAYERS))
                                            (PILES (PLAYERS))
                                            (SET DECK POT HOLE))
|| 4:17                           --- [EVAL by RULE346] --->
                                  T

                            (SHOW T)
|< 4:18                     --- [SUCCEED by RULE34] --->
               (CONSIDER (UNION (UNION (PILES (PLAYERS)) (SET DECK POT HOLE))
                       (INTERSECT (UNION (HANDS (PLAYERS)) (PILES (PLAYERS)) (SET DECK POT HOLE))
                               (SET (HAND ME)))))

                       (INTERSECT (UNION (HANDS (PLAYERS)) (PILES (PLAYERS)) (SET DECK POT HOLE))
                               (SET (HAND ME)))
|> 4:19                --- [by RULE268] --->
                (CONSIDER (INTERSECT (UNION (HANDS (PLAYERS)) (PILES (PLAYERS)) (SET DECK POT HOLE))
                               (SET (HAND ME))))

                       (INTERSECT (UNION (HANDS (PLAYERS)) (PILES (PLAYERS)) (SET DECK POT HOLE))
                               (SET (HAND ME)))
|| 4:20                --- [UNBRACKET by RULE166] --->
                       (IF (IN (HAND ME)
                             (UNION (HANDS (PLAYERS)) (PILES (PLAYERS)) (SET DECK POT HOLE)))
                          (SET (HAND ME))
                          NIL)

                          (IN (HAND ME)
                             (UNION (HANDS (PLAYERS)) (PILES (PLAYERS)) (SET DECK POT HOLE)))
|| 4:21                   --- [DISTRIBUTE by RULE284] --->
                          (OR [IN (HAND ME) (HANDS (PLAYERS))]
                             [IN (HAND ME) (PILES (PLAYERS))]
                             [IN (HAND ME) (SET DECK POT HOLE)])

                                (HANDS (PLAYERS))
|| 4:22                         --- [ELABORATE by RULE124] --->
                                (PROJECT HAND (PLAYERS))

                             (IN (HAND ME) (PROJECT HAND (PLAYERS)))
|| 4:23                      --- [REMOVE-QUANT by RULE172] --->
                             (IN ME (PLAYERS))

|| 4:24                      --- [EVAL by RULE173] --->
                             T

                          (OR T [IN (HAND ME) (PILES (PLAYERS))]
                             [IN (HAND ME) (SET DECK POT HOLE)])
|| 4:25                   --- [EVAL by RULE184] --->
                          T

                       (IF T (SET (HAND ME)) NIL)
|| 4:26                --- [SIMPLIFY by RULE185] --->
                       (SET (HAND ME))

                (CONSIDER (SET (HAND ME)))
|< 4:27         --- [by RULE269] --->
           (CONSIDER (UNION (UNION (PILES (PLAYERS)) (SET DECK POT HOLE))
                               (SET (HAND ME))))

                  (UNION (UNION (PILES (PLAYERS)) (SET DECK POT HOLE))
                       (SET (HAND ME)))
| 4:28            --- [SIMPLIFY by RULE177] --->
                  (UNION (PILES (PLAYERS)) (SET DECK POT HOLE) (SET (HAND ME)))

| 4:29            --- [SIMPLIFY by RULE187] --->
                  (UNION (SET DECK POT HOLE (HAND ME)) (PILES (PLAYERS)))

           (CONSIDER (UNION (SET DECK POT HOLE (HAND ME)) (PILES (PLAYERS))))
< 4:30     --- [by RULE269] --->
       (EVAL (NOT (IN (LOC QS)
                  (UNION (SET DECK POT HOLE (HAND ME)) (PILES (PLAYERS))))))

               (IN (LOC QS)
                  (UNION (SET DECK POT HOLE (HAND ME)) (PILES (PLAYERS))))
 4:31          --- [DISTRIBUTE by RULE284] --->
               (OR [IN (LOC QS) (SET DECK POT HOLE (HAND ME))]
                  [IN (LOC QS) (PILES (PLAYERS))]])

                  (IN (LOC QS) (SET DECK POT HOLE (HAND ME)))
 4:32             --- [DISTRIBUTE by RULE182] --->
                  (OR [* (LOC QS) DECK]
                     [* (LOC QS) POT]
                     [* (LOC QS) HOLE]
                     [* (LOC QS) (HAND ME)]])

               (OR [OR [* (LOC QS) DECK]
                     [* (LOC QS) POT]
                     [* (LOC QS) HOLE]
                     [* (LOC QS) (HAND ME)]]
                  [IN (LOC QS) (PILES (PLAYERS))]])
 4:33          --- [SIMPLIFY by RULE177] --->
```

```
                    (OR [= (LOC QS) DECK]
                        [= (LOC QS) POT]
                        [= (LOC QS) HOLE]
                        [= (LOC QS) (HAND ME)]
                        [IN (LOC QS) (PILES (PLAYERS))])

                        (= (LOC QS) DECK)
4:34                    --- [RECOGNIZE by RULE123] --->
                        (AT QS DECK)

                        (= (LOC QS) POT)
4:35                    --- [RECOGNIZE by RULE123] --->
                        (AT QS POT)

                        (= (LOC QS) HOLE)
4:36                    --- [RECOGNIZE by RULE123] --->
                        (AT QS HOLE)

                        (= (LOC QS) (HAND ME))
4:37                    --- [RECOGNIZE by RULE123] --->
                        (AT QS (HAND ME))

                        (AT QS DECK)
4:38                    --- [EVAL by RULE180] --->
                        NIL

                    (OR NIL [AT QS POT]
                        [AT QS HOLE]
                        [AT QS (HAND ME)]
                        [IN (LOC QS) (PILES (PLAYERS))])
4:39                --- [SIMPLIFY by RULE178] --->
                    (OR [AT QS POT]
                        [AT QS HOLE]
                        [AT QS (HAND ME)]
                        [IN (LOC QS) (PILES (PLAYERS))])

                        (AT QS POT)
4:40                    --- [RECOGNIZE by RULE123] --->
                        (IN-POT QS)

                        (AT QS (HAND ME))
4:41                    --- [RECOGNIZE by RULE123] --->
                        (HAS ME QS)

4:42                    --- [RECOGNIZE by RULE123] --->
                        (HAS-ME QS)

                        (IN (LOC QS) (PILES (PLAYERS)))
> 4:43                  --- [by RULE268] --->
                (CONSIDER (IN (LOC QS) (PILES (PLAYERS))))

                            (PILES (PLAYERS))
| 4:44                      --- [ELABORATE by RULE124] --->
                            (PROJECT PILE (PLAYERS))

                        (IN (LOC QS) (PROJECT PILE (PLAYERS)))
| 4:45                  --- [by RULE14] --->
                        (EXISTS P3 (PLAYERS) (= (LOC QS) (PILE P3)))

                                (= (LOC QS) (PILE P3))
| 4:46                          --- [RECOGNIZE by RULE123] --->
                                (AT QS (PILE P3))

| 4:47                          --- [by RULE356] --->
                                (OR [WAS-DURING (CURRENT ROUND-IN-PROGRESS)
                                                (CAUSE (AT QS (PILE P3)))]
                                    [BEFORE (CURRENT ROUND-IN-PROGRESS) (AT QS (PILE P3))])

                                    (BEFORE (CURRENT ROUND-IN-PROGRESS) (AT QS (PILE P3)))
| 4:48                              --- [EVAL by RULE367] --->
                                    NIL

                                (OR [WAS-DURING (CURRENT ROUND-IN-PROGRESS)
                                                (CAUSE (AT QS (PILE P3)))]
                                    NIL)
| 4:49                          --- [SIMPLIFY by RULE178] --->
                                (OR [WAS-DURING (CURRENT ROUND-IN-PROGRESS)
                                                (CAUSE (AT QS (PILE P3)))])

| 4:50                          --- [SIMPLIFY by RULE178] --->
                                (WAS-DURING (CURRENT ROUND-IN-PROGRESS)
                                             (CAUSE (AT QS (PILE P3))))

                                             (CAUSE (AT QS (PILE P3)))
| 4:51                                       --- [RECOGNIZE by RULE358] --->
                                             (MOVE QS LOC1 (PILE P3))

| 4:52                                       --- [RECOGNIZE by RULE123] --->
                                             (TAKE P3 QS)

                                 (WAS-DURING (CURRENT ROUND-IN-PROGRESS) (TAKE P3 QS))
| 4:53                           --- [RECOGNIZE by RULE123] --->
                                 (TOOK P3 QS)

                         (EXISTS P3 (PLAYERS) (TOOK P3 QS))
| 4:54                   --- [RECOGNIZE by RULE123] --->
                         (TAKEN QS)

               (CONSIDER (TAKEN QS))
< 4:55         --- [by RULE269] --->
          (EVAL (NOT (OR [IN-POT QS] [AT QS HOLE] [HAS-ME QS] [TAKEN QS])))

          [Final expression:]
          (EVAL (NOT (OR [IN-POT QS] [AT QS HOLE] [HAS-ME QS] [TAKEN QS])))

(ASSUMING (DISJOINT (HANDS (PLAYERS))
                    (PILES (PLAYERS))
                    (SET DECK POT HOLE)))
NIL
<43>recordfile

RECORD FILE DSK: (DV4D07 . PZG) CLOSED 07-DEC-80 18:25:03
```

D.5. DERIV5: Flush the Queen without Taking it

```
RECORD FILE DSK: (DV5D07 . PZG) OPENED 07-DEC-80 18:25:31
NIL
<83>p-links nil

(DERIV5 STARTED "14-APR-80 22:42:30" --DOMAIN: HEARTS --PROBLEM:
        (AND [ACHIEVE (FLUSH QS)] [AVOID (TAKE ME QS) (TRICK)]))


          [Initial expression:]
          (AND [ACHIEVE (FLUSH QS)] [AVOID (TAKE ME QS) (TRICK)])

               (ACHIEVE (FLUSH QS))
 5:1           --- [REDUCE by RULE301] --->
               (UNTIL (PLAYED! QS) (ACHIEVE (LEAD-SPADE ME)))

               (AVOID (TAKE ME QS) (TRICK))
 5:2           --- [ELABORATE by RULE124] --->
               (ACHIEVE (NOT (DURING (TRICK) (TAKE ME QS))))

          (AND [UNTIL (PLAYED! QS) (ACHIEVE (LEAD-SPADE ME))]
               [ACHIEVE (NOT (DURING (TRICK) (TAKE ME QS)))])
 5:3      --- [by RULE359] --->
          (UNTIL (PLAYED! QS)
                 (AND [ACHIEVE (LEAD-SPADE ME)]
                      [ACHIEVE (NOT (DURING (TRICK) (TAKE ME QS)))]))

                 (AND [ACHIEVE (LEAD-SPADE ME)]
                      [ACHIEVE (NOT (DURING (TRICK) (TAKE ME QS)))])
 5:4             --- [COLLECT by RULE360] --->
                 (ACHIEVE (AND [LEAD-SPADE ME] [NOT (DURING (TRICK) (TAKE ME QS))]))

                                    (DURING (TRICK) (TAKE ME QS))
> 5:5                               --- [by RULE268] --->
                          (CONSIDER (DURING (TRICK) (TAKE ME QS)))

                                            (TRICK)
| 5:6                                       --- [ELABORATE by RULE124] --->
                                            (SCENARIO (EACH P1 (PLAYERS) (PLAY-CARD P1))
                                                      (TAKE-TRICK (TRICK-WINNER)))
```

```
                    (DURING (SCENARIO (EACH P1 (PLAYERS) (PLAY-CARD P1))
                                      (TAKE-TRICK (TRICK-WINNER)))
                            (TAKE ME QS))
| 5:7               --- [DISTRIBUTE by RULE284] --->
                    (OR [DURING (EACH P1 (PLAYERS) (PLAY-CARD P1)) (TAKE ME QS)]
                        [DURING (TAKE-TRICK (TRICK-WINNER)) (TAKE ME QS)])

                        (DURING (EACH P1 (PLAYERS) (PLAY-CARD P1)) (TAKE ME QS))
| 5:8                   --- [COMPUTE by RULE57] --->
                        NIL

                    (OR NIL [DURING (TAKE-TRICK (TRICK-WINNER)) (TAKE ME QS)])
| 5:9               --- [SIMPLIFY by RULE341] --->
                    (DURING (TAKE-TRICK (TRICK-WINNER)) (TAKE ME QS))

                            (TAKE-TRICK (TRICK-WINNER))
| 5:10                      --- [ELABORATE by RULE124] --->
                            (EACH C1 (CARDS-PLAYED) (TAKE (TRICK-WINNER) C1))

                    (DURING (EACH C1 (CARDS-PLAYED) (TAKE (TRICK-WINNER) C1))
                            (TAKE ME QS))
| 5:11              --- [COLLECT by RULE58] --->
                    (EXISTS C1 (CARDS-PLAYED)
                            (DURING (TAKE (TRICK-WINNER) C1) (TAKE ME QS)))

                            (DURING (TAKE (TRICK-WINNER) C1) (TAKE ME QS))
| 5:12                      --- [DISTRIBUTE by RULE43] --->
                            (AND [= (TRICK-WINNER) ME] [= C1 QS])

                    (EXISTS C1 (CARDS-PLAYED) (AND [= (TRICK-WINNER) ME] [= C1 QS]))
| 5:13              --- [REMOVE-QUANT by RULE59] --->
                    (AND [IN QS (CARDS-PLAYED)] [= (TRICK-WINNER) ME])

                            (TRICK-WINNER)
| 5:14                      --- [ELABORATE by RULE124] --->
                            (PLAYER-OF (WINNING-CARD))

                        (= (PLAYER-OF (WINNING-CARD)) ME)
| 5:15                  --- [LEMMA by RULE146] --->
                        (= (CARD-OF ME) (WINNING-CARD))

                            (WINNING-CARD)
| 5:16                      --- [ELABORATE by RULE124] --->
                            (HIGHEST-IN-SUIT-LED (CARDS-PLAYED))

| 5:17                      --- [ELABORATE by RULE124] --->
                            (HIGHEST (CARDS-IN-SUIT-LED (CARDS-PLAYED)))

| 5:18                      --- [ELABORATE by RULE124] --->
                            (THE C2 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
                                 (FORALL X1 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
                                         (NOT (HIGHER X1 C2))))

                        (= (CARD-OF ME)
                           (THE C2 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
                                (FORALL X1 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
                                        (NOT (HIGHER X1 C2)))))
| 5:19                  --- [UNBRACKET by RULE64] --->
                        (AND [IN (CARD-OF ME) (CARDS-IN-SUIT-LED (CARDS-PLAYED))]
                             [FORALL X1 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
                                     (NOT (HIGHER X1 (CARD-OF ME)))])

                                (CARDS-IN-SUIT-LED (CARDS-PLAYED))
| 5:20                          --- [ELABORATE by RULE124] --->
                                (FILTER (CARDS-PLAYED) IN-SUIT-LED)

                              (IN (CARD-OF ME) (FILTER (CARDS-PLAYED) IN-SUIT-LED))
| 5:21                        --- [ELABORATE by RULE361] --->
                              (AND [IN (CARD-OF ME) (CARDS-PLAYED)] [IN-SUIT-LED (CARD-OF ME)])

                                   (CARDS-PLAYED)
| 5:22                             --- [ELABORATE by RULE124] --->
                                   (PROJECT CARD-OF (PLAYERS))

                                 (IN (CARD-OF ME) (PROJECT CARD-OF (PLAYERS)))
| 5:23                           --- [REMOVE-QUANT by RULE172] --->
                                 (IN ME (PLAYERS))

| 5:24                           --- [EVAL by RULE173] --->
                                 T
```

```
                                            (AND T [IN-SUIT-LED (CARD-OF ME)])
| 5:25                                      --- [SIMPLIFY by RULE341] --->
                                            (IN-SUIT-LED (CARD-OF ME))

| 5:26                                      --- [ELABORATE by RULE124] --->
                                            (= (SUIT-OF (CARD-OF ME)) (SUIT-LED))

                                               (SUIT-LED)
| 5:27                                         --- [ELABORATE by RULE124] --->
                                               (SUIT-OF (CARD-OF (LEADER)))

                         (CONSIDER (AND [IN QS (CARDS-PLAYED)]
                                       [AND [= (SUIT-OF (CARD-OF ME)) (SUIT-OF (CARD-OF (LEADER)))]
                                            [FORALL X1 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
                                                   (NOT (HIGHER X1 (CARD-OF ME)))]]))
< 5:28                   --- [by RULE269] --->
         (UNTIL (PLAYED! QS)
                (ACHIEVE (AND [LEAD-SPADE ME]
                              [NOT (AND [IN QS (CARDS-PLAYED)]
                                       [AND [= (SUIT-OF (CARD-OF ME)) (SUIT-OF (CARD-OF (LEADER)))]
                                            [FORALL X1 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
                                                   (NOT (HIGHER X1 (CARD-OF ME)))]])]))))

                              (LEAD-SPADE ME)
 5:29                         --- [ELABORATE by RULE124] --->
                              (AND [LEADING ME] [SPADE! (CARD-OF ME)])

                                  (LEADING ME)
 5:30                             --- [ELABORATE by RULE124] --->
                                  (= (LEADER) ME)

                         (AND [AND [= (LEADER) ME] [SPADE! (CARD-OF ME)]]
                              [NOT (AND [IN QS (CARDS-PLAYED)]
                                       [AND [= (SUIT-OF (CARD-OF ME)) (SUIT-OF (CARD-OF (LEADER)))]
                                            [FORALL X1 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
                                                   (NOT (HIGHER X1 (CARD-OF ME)))]])])
 5:31                    --- [SIMPLIFY by RULE177] --->
                         (AND [= (LEADER) ME]
                              [SPADE! (CARD-OF ME)]
                              [NOT (AND [IN QS (CARDS-PLAYED)]
                                       [AND [= (SUIT-OF (CARD-OF ME)) (SUIT-OF (CARD-OF (LEADER)))]
                                            [FORALL X1 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
                                                   (NOT (HIGHER X1 (CARD-OF ME)))]])])

 5:32                    --- [by RULE354] --->
                         (AND [= (LEADER) ME]
                              [SPADE! (CARD-OF ME)]
                              [NOT (AND [IN QS (CARDS-PLAYED)]
                                       [AND [= (SUIT-OF (CARD-OF ME)) (SUIT-OF (CARD-OF ME))]
                                            [FORALL X1 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
                                                   (NOT (HIGHER X1 (CARD-OF ME)))]])])

                                            (= (SUIT-OF (CARD-OF ME)) (SUIT-OF (CARD-OF ME)))
 5:33                                       --- [EVAL by RULE179] --->
                                            T

                                       (AND T
                                            [FORALL X1 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
                                                   (NOT (HIGHER X1 (CARD-OF ME)))])
 5:34                                  --- [SIMPLIFY by RULE341] --->
                                       (FORALL X1 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
                                               (NOT (HIGHER X1 (CARD-OF ME))))

                              (NOT (AND [IN QS (CARDS-PLAYED)]
                                       [FORALL X1 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
                                               (NOT (HIGHER X1 (CARD-OF ME)))]))
 5:35                         --- [by RULE109] --->
                              (=> (AND [IN QS (CARDS-PLAYED)])
                                  (NOT (NOT (HIGHER (FIND-ELT (CARDS-IN-SUIT-LED (CARDS-PLAYED)))
                                                    (CARD-OF ME)))))

> 5:36                        --- [by RULE268] --->
                    (CONSIDER (=> (AND [IN QS (CARDS-PLAYED)])
                                  (NOT (NOT (HIGHER (FIND-ELT (CARDS-IN-SUIT-LED (CARDS-PLAYED)))
                                                    (CARD-OF ME))))))

                                  (AND [IN QS (CARDS-PLAYED)])
| 5:37                            --- [SIMPLIFY by RULE178] --->
                                  (IN QS (CARDS-PLAYED))

                                  (NOT (NOT (HIGHER (FIND-ELT (CARDS-IN-SUIT-LED (CARDS-PLAYED)))
                                                 (CARD-OF ME))))
| 5:38                               --- [SIMPLIFY by RULE88] --->
                                     (HIGHER (FIND-ELT (CARDS-IN-SUIT-LED (CARDS-PLAYED)))
                                             (CARD-OF ME))

                                             (FIND-ELT (CARDS-IN-SUIT-LED (CARDS-PLAYED)))
|> 5:39                                      --- [by RULE84] --->
                                       (SHOW (IN QS (CARDS-IN-SUIT-LED (CARDS-PLAYED))))

                                                 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
|| 5:40                                          --- [ELABORATE by RULE124] --->
                                                 (FILTER (CARDS-PLAYED) IN-SUIT-LED)

                                             (IN QS (FILTER (CARDS-PLAYED) IN-SUIT-LED))
|| 5:41                                      --- [ELABORATE by RULE361] --->
                                             (AND [IN QS (CARDS-PLAYED)] [IN-SUIT-LED QS])

                                                  (IN QS (CARDS-PLAYED))
|| 5:42                                           --- [REDUCE by RULE301] --->
                                                  T

                                             (AND T [IN-SUIT-LED QS])
|| 5:43                                      --- [SIMPLIFY by RULE341] --->
                                             (IN-SUIT-LED QS)

|| 5:44                                      --- [ELABORATE by RULE124] --->
                                             (= (SUIT-OF QS) (SUIT-LED))

                                                (SUIT-OF QS)
|| 5:45                                         --- [EVAL by RULE2] --->
                                                S

                                                (SUIT-LED)
|| 5:46                                         --- [REDUCE by RULE301] --->
                                                (SUIT-OF (CARD-OF ME))

|| 5:47                                         --- [REDUCE by RULE301] --->
                                                S

                                             (= S S)
|| 5:48                                      --- [EVAL by RULE179] --->
                                             T

                                         (SHOW T)
|< 5:49                                  --- [SUCCEED by RULE34] --->
                              (CONSIDER (=> (IN QS (CARDS-PLAYED)) (HIGHER QS (CARD-OF ME))))

                                       (=> (IN QS (CARDS-PLAYED)) (HIGHER QS (CARD-OF ME)))
| 5:50                                 --- [by RULE157] --->
                                       (HIGHER QS (CARD-OF ME))

                              (CONSIDER (HIGHER QS (CARD-OF ME)))
< 5:51                        --- [by RULE269] --->
                (UNTIL (PLAYED! QS)
                       (ACHIEVE (AND [= (LEADER) ME]
                                     [SPADE! (CARD-OF ME)]
                                     [HIGHER QS (CARD-OF ME)])))

                                       (= (LEADER) ME)
 5:52                                  --- [RECOGNIZE by RULE123] --->
                                       (LEADING ME)

                                  (AND [LEADING ME] [SPADE! (CARD-OF ME)] [HIGHER QS (CARD-OF ME)])
 5:53                             --- [RECOGNIZE by RULE123] --->
                                  (LEAD-SAFE-SPADE ME)

        [Final expression:]
        (UNTIL (PLAYED! QS) (ACHIEVE (LEAD-SAFE-SPADE ME)))
NIL
<64>recordfile

RECORD FILE DSK: (DV5D07 . PZG) CLOSED 07-DEC-80 18:29:02
```

D.6. DERIV6: Avoid Taking Points (Play a Low Card)

```
RECORD FILE DSK: (DV6D07 . PZG) OPENED 07-DEC-80 18:29:26
NIL
<72>p-links nil

(DERIV6 STARTED "21-APR-80 18:33:13" --DOMAIN: HEARTS --PROBLEM:
        (AVOID-TAKING-POINTS (TRICK)))


          [Initial expression:]
          (AVOID-TAKING-POINTS (TRICK))

 6:1      --- [ELABORATE by RULE124] --->
          (AVOID (TAKE-POINTS ME) (TRICK))

 6:2      --- [ELABORATE by RULE124] --->
          (ACHIEVE (NOT (DURING (TRICK) (TAKE-POINTS ME))))

                                (TRICK)
 6:3                            --- [ELABORATE by RULE124] --->
                                (SCENARIO (EACH P1 (PLAYERS) (PLAY-CARD P1))
                                          (TAKE-TRICK (TRICK-WINNER)))

                        (DURING (SCENARIO (EACH P1 (PLAYERS) (PLAY-CARD P1))
                                          (TAKE-TRICK (TRICK-WINNER)))
                                (TAKE-POINTS ME))
 6:4                    --- [DISTRIBUTE by RULE284] --->
                        (OR [DURING (EACH P1 (PLAYERS) (PLAY-CARD P1)) (TAKE-POINTS ME)]
                            [DURING (TAKE-TRICK (TRICK-WINNER)) (TAKE-POINTS ME)])

                            (DURING (EACH P1 (PLAYERS) (PLAY-CARD P1)) (TAKE-POINTS ME))
 6:5                        --- [COMPUTE by RULE57] --->
                            NIL

                        (OR NIL [DURING (TAKE-TRICK (TRICK-WINNER)) (TAKE-POINTS ME)])
 6:6                    --- [SIMPLIFY by RULE341] --->
                        (DURING (TAKE-TRICK (TRICK-WINNER)) (TAKE-POINTS ME))

                                (TAKE-POINTS ME)
 6:7                            --- [ELABORATE by RULE124] --->
                                (FOR-SOME C1 (POINT-CARDS) (TAKE ME C1))

                                (TAKE-TRICK (TRICK-WINNER))
 6:8                            --- [ELABORATE by RULE124] --->
                                (EACH C3 (CARDS-PLAYED) (TAKE (TRICK-WINNER) C3))

                        (DURING (EACH C3 (CARDS-PLAYED) (TAKE (TRICK-WINNER) C3))
                                (FOR-SOME C1 (POINT-CARDS) (TAKE ME C1)))
 6:9                    --- [COLLECT by RULE58] --->
                        (EXISTS C3 (CARDS-PLAYED)
                                (DURING (TAKE (TRICK-WINNER) C3)
                                        (FOR-SOME C1 (POINT-CARDS) (TAKE ME C1))))

                                (DURING (TAKE (TRICK-WINNER) C3)
                                        (FOR-SOME C1 (POINT-CARDS) (TAKE ME C1)))
 6:10                           --- [COLLECT by RULE100] --->
                                (EXISTS C1 (POINT-CARDS)
                                        (DURING (TAKE (TRICK-WINNER) C3) (TAKE ME C1)))

                                        (DURING (TAKE (TRICK-WINNER) C3) (TAKE ME C1))
 6:11                                   --- [DISTRIBUTE by RULE43] --->
                                        (AND [= (TRICK-WINNER) ME] [= C3 C1])

                                (EXISTS C1 (POINT-CARDS) (AND [= (TRICK-WINNER) ME] [= C3 C1]))
 6:12                           --- [REMOVE-QUANT by RULE69] --->
                                (AND [IN C3 (POINT-CARDS)] [= (TRICK-WINNER) ME])

                        (EXISTS C3 (CARDS-PLAYED)
                                (AND [IN C3 (POINT-CARDS)] [= (TRICK-WINNER) ME]))
 6:13                   --- [SIMPLIFY-QUANT by RULE108] --->
                        (AND [= (TRICK-WINNER) ME]
                             [EXISTS C3 (CARDS-PLAYED) (AND [IN C3 (POINT-CARDS)])])

                                     (AND [IN C3 (POINT-CARDS)])
 6:14                                --- [SIMPLIFY by RULE178] --->
                                     (IN C3 (POINT-CARDS))
```

```
                                          (POINT-CARDS)
6:15                                      --- [ELABORATE by RULE124] --->
                                          (SET-OF C4 (CARDS) (HAS-POINTS C4))

                                      (IN C3 (SET-OF C4 (CARDS) (HAS-POINTS C4)))
6:16                                  --- [REMOVE-QUANT by RULE10] --->
                                      (AND [IN C3 (CARDS)] [HAS-POINTS C3])

                              (EXISTS C3 (CARDS-PLAYED) (AND [IN C3 (CARDS)] [HAS-POINTS C3]))
> 6:17                        --- [SIMPLIFY-QUANT by RULE362] --->
                        (SHOW (SUBSET (CARDS-PLAYED) (CARDS)))

                              (SUBSET (CARDS-PLAYED) (CARDS))
| 6:18                        --- [FACT by RULE363] --->
                              T

                        (SHOW T)
< 6:19                  --- [SUCCEED by RULE34] --->
            (ACHIEVE (NOT (AND [= (TRICK-WINNER) ME]
                              [EXISTS C3 (CARDS-PLAYED) (AND [HAS-POINTS C3])])))

                                      (AND [HAS-POINTS C3])
6:20                                  --- [SIMPLIFY by RULE178] --->
                                      (HAS-POINTS C3)

                              (EXISTS C3 (CARDS-PLAYED) (HAS-POINTS C3))
6:21                          --- [RECOGNIZE by RULE123] --->
                              (HAVE-POINTS (CARDS-PLAYED))

6:22                          --- [RECOGNIZE by RULE123] --->
                              (TRICK-HAS-POINTS)

                              (= (TRICK-WINNER) ME)
> 6:23                        --- [by RULE268] --->
                    (CONSIDER (= (TRICK-WINNER) ME))

                                 (TRICK-WINNER)
| 6:24                           --- [ELABORATE by RULE124] --->
                                 (PLAYER-OF (WINNING-CARD))

| 6:25                           --- [ELABORATE by RULE124] --->
                                 (THE P2 (PLAYERS) (= (CARD-OF P2) (WINNING-CARD)))

                              (= (THE P2 (PLAYERS) (= (CARD-OF P2) (WINNING-CARD))) ME)
| 6:26                        --- [REMOVE-QUANT by RULE131] --->
                              (AND [IN ME (PLAYERS)] [= (CARD-OF ME) (WINNING-CARD)])

                                   (IN ME (PLAYERS))
| 6:27                             --- [FACT by RULE173] --->
                                   T

                              (AND T [= (CARD-OF ME) (WINNING-CARD)])
| 6:28                        --- [SIMPLIFY by RULE341] --->
                              (= (CARD-OF ME) (WINNING-CARD))

                                 (WINNING-CARD)
| 6:29                           --- [ELABORATE by RULE124] --->
                                 (HIGHEST-IN-SUIT-LED (CARDS-PLAYED))

| 6:30                           --- [ELABORATE by RULE124] --->
                                 (HIGHEST (CARDS-IN-SUIT-LED (CARDS-PLAYED)))

| 6:31                           --- [ELABORATE by RULE124] --->
                                 (THE C5 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
                                      (FORALL X1 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
                                              (NOT (HIGHER X1 C5))))

                              (= (CARD-OF ME)
                                 (THE C5 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
                                      (FORALL X1 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
                                              (NOT (HIGHER X1 C5)))))
| 6:32                        --- [REMOVE-QUANT by RULE64] --->
                              (AND [IN (CARD-OF ME) (CARDS-IN-SUIT-LED (CARDS-PLAYED))]
                                   [FORALL X1 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
                                           (NOT (HIGHER X1 (CARD-OF ME)))])

                                   (IN (CARD-OF ME) (CARDS-IN-SUIT-LED (CARDS-PLAYED)))
| 6:33                             --- [by RULE149] --->
                                   (AND [IN-SUIT-LED (CARD-OF ME)] [IN (CARD-OF ME) (CARDS-PLAYED)])
```

```
                                            (CARDS-PLAYED)
| 6:34                                      --- [ELABORATE by RULE124] --->
                                            (PROJECT CARD-OF (PLAYERS))

                                        (IN (CARD-OF ME) (PROJECT CARD-OF (PLAYERS)))
| 6:35                                  --- [REMOVE-QUANT by RULE172] --->
                                        (IN ME (PLAYERS))

| 6:36                                  --- [FACT by RULE173] --->
                                        T

                                   (AND [IN-SUIT-LED (CARD-OF ME)] T)
| 6:37                             --- [SIMPLIFY by RULE343] --->
                                   (IN-SUIT-LED (CARD-OF ME))

                   (CONSIDER (AND [IN-SUIT-LED (CARD-OF ME)]
                                  [FORALL X1 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
                                          (NOT (HIGHER X1 (CARD-OF ME)))]]))
< 6:38             --- [by RULE269] --->
          (ACHIEVE (NOT (AND [AND [IN-SUIT-LED (CARD-OF ME)]
                                  [FORALL X1 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
                                          (NOT (HIGHER X1 (CARD-OF ME)))]]
                             [TRICK-HAS-POINTS])))

                        (AND [AND [IN-SUIT-LED (CARD-OF ME)]
                                  [FORALL X1 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
                                          (NOT (HIGHER X1 (CARD-OF ME)))]]
                             [TRICK-HAS-POINTS])
 6:39                   --- [SIMPLIFY by RULE177] --->
                        (AND [IN-SUIT-LED (CARD-OF ME)]
                             [FORALL X1 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
                                     (NOT (HIGHER X1 (CARD-OF ME)))]
                             [TRICK-HAS-POINTS])

                   (NOT (AND [IN-SUIT-LED (CARD-OF ME)]
                             [FORALL X1 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
                                     (NOT (HIGHER X1 (CARD-OF ME)))]
                             [TRICK-HAS-POINTS]))
 6:40              --- [by RULE109] --->
                   (=> (AND [IN-SUIT-LED (CARD-OF ME)] [TRICK-HAS-POINTS])
                       (NOT (NOT (HIGHER (FIND-ELT (CARDS-IN-SUIT-LED (CARDS-PLAYED)))
                                         (CARD-OF ME)))))

                       (NOT (NOT (HIGHER (FIND-ELT (CARDS-IN-SUIT-LED (CARDS-PLAYED)))
                                         (CARD-OF ME))))
 6:41                  --- [SIMPLIFY by RULE88] --->
                       (HIGHER (FIND-ELT (CARDS-IN-SUIT-LED (CARDS-PLAYED)))
                               (CARD-OF ME))

> 6:44                 --- [TRY-THIS by RULE142] --->
                  (SHOW (AND [IN CARD1 (CARDS-IN-SUIT-LED (CARDS-PLAYED))]
                             [HIGHER CARD1 (CARD-OF ME)]))

                            (IN CARD1 (CARDS-IN-SUIT-LED (CARDS-PLAYED)))
| 6:45                      --- [by RULE149] --->
                            (AND [IN-SUIT-LED CARD1] [IN CARD1 (CARDS-PLAYED)])

                                     (CARDS-PLAYED)
| 6:46                               --- [ELABORATE by RULE124] --->
                                     (PROJECT CARD-OF (PLAYERS))

                                 (IN CARD1 (PROJECT CARD-OF (PLAYERS)))
|> 6:47                          --- [by RULE364] --->
                           (SHOW (IN (LEADER) (PLAYERS)))
(CARD1 <- BINDING : OK)

                                 (IN (LEADER) (PLAYERS))
|| 6:48                          --- [FACT by RULE365] --->
                                 T

                           (SHOW T)
|< 6:49                    --- [SUCCEED by RULE34] --->
                  (SHOW (AND [AND [IN-SUIT-LED CARD1] T] [HIGHER CARD1 (CARD-OF ME)]))

                            (AND [IN-SUIT-LED CARD1] T)
| 6:50                      --- [SIMPLIFY by RULE343] --->
                            (IN-SUIT-LED CARD1)

                                         CARD1
| 6:51                                   --- [EVAL by RULE127] --->
                                         OK
```

```
                              (IN-SUIT-LED DK)
| 6:52                        --- [ELABORATE by RULE124] --->
                              (= (SUIT-OF DK) (SUIT-LED))

                                 (SUIT-LED)
| 6:53                           --- [ELABORATE by RULE124] --->
                                 (SUIT-OF (CARD-OF (LEADER)))

                              (= (SUIT-OF DK) (SUIT-OF (CARD-OF (LEADER))))
|> 6:54                       --- [by RULE91] --->
                        (SHOW (AND [= DK (CARD-OF (LEADER))]))

                                      (CARD-OF (LEADER))
|| 6:55                               --- [EVAL by RULE346] --->
                                      DK

                                   (= DK DK)
|| 6:56                            --- [EVAL by RULE179] --->
                                   T

                              (AND T)
|| 6:57                       --- [COMPUTE by RULE236] --->
                              T

                        (SHOW T)
|< 6:58                 --- [SUCCEED by RULE53] --->
                   (SHOW (AND T [HIGHER CARD1 (CARD-OF ME)]))

                         (AND T [HIGHER CARD1 (CARD-OF ME)])
| 6:59                   --- [SIMPLIFY by RULE341] --->
                         (HIGHER CARD1 (CARD-OF ME))

                                 CARD1
| 6:60                           --- [EVAL by RULE127] --->
                                 DK

                   (SHOW (HIGHER DK (CARD-OF ME)))
< 6:61             --- [GIVE-UP by RULE151] --->
             (ACHIEVE (=> (AND [IN-SUIT-LED (CARD-OF ME)] [TRICK-HAS-POINTS])
                         (HIGHER DK (CARD-OF ME))))

                         (HIGHER DK (CARD-OF ME))
 6:62                    --- [by RULE153] --->
                         (LOWER (CARD-OF ME) DK)

             [Final expression:]
             (ACHIEVE (=> (AND [IN-SUIT-LED (CARD-OF ME)] [TRICK-HAS-POINTS])
                         (LOWER (CARD-OF ME) DK)))

(ASSUMING (CARD-OF (LEADER)) -> DK)
NIL
<73>recordfile

RECORD FILE DSK: (DV6D07 . PZG) CLOSED 07-DEC-80 18:32:12
```

D.7. DERIV7: Get the Lead

```
RECORD FILE DSK: (DV7D07 . PZG) OPENED 07-DEC-80 14:26:11
NIL
<100>p-links nil

(DERIV7 STARTED "07-DEC-80 14: 7:50" --DOMAIN: HEARTS --PROBLEM:
        (ACHIEVE (LEADING ME)))

         [Initial expression:]
         (ACHIEVE (LEADING ME))

                    (LEADING ME)
 7:1                --- [ELABORATE by RULE124] --->
                    (= (LEADER) ME)

                       (LEADER)
 7:2                   --- [ELABORATE by RULE124] --->
```

```
                    (PREVIOUS (TRICK-WINNER))

         (ACHIEVE (= (PREVIOUS (TRICK-WINNER)) ME))
 7:3     --- [by RULE366] --->
         (ACHIEVE (= (TRICK-WINNER) ME))

                 (= (TRICK-WINNER) ME)
 7:4             --- [REDUCE by RULE301] --->
                 (AND [IN-SUIT-LED (CARD-OF ME)]
                      [FORALL X1 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
                              (NOT (HIGHER X1 (CARD-OF ME)))])

                                  (HIGHER X1 (CARD-OF ME))
 7:5                              --- [by RULE153] --->
                                  (LOWER (CARD-OF ME) X1)

 7:6                              --- [REDUCE by RULE154] --->
                                  (LOW (CARD-OF ME))

                      (FORALL X1 (CARDS-IN-SUIT-LED (CARDS-PLAYED))
                              (NOT (LOW (CARD-OF ME))))
 7:7                  --- [REMOVE-QUANT by RULE155] --->
                      (NOT (LOW (CARD-OF ME)))

 7:8                  --- [by RULE367] --->
                      (HIGH (CARD-OF ME))

         [Final expression:]
         (ACHIEVE (AND [IN-SUIT-LED (CARD-OF ME)] [HIGH (CARD-OF ME)]))
<1>recordfile

RECORD FILE DSK: (DV7D07 . PZG) CLOSED 07-DEC-80 14:26:41
```

D.8. DERIV8: Get Void

```
RECORD FILE DSK: (DV8D07 . PZG) OPENED 07-DEC-80 14:33:26
NIL
<41>p-links nil

(DERIV8 STARTED "07-DEC-80 14:27:34" --DOMAIN: HEARTS --PROBLEM:
        (ACHIEVE (VOID ME S0)))


         [Initial expression:]
         (ACHIEVE (VOID ME S0))

                 (VOID ME S0)
 8:1             --- [ELABORATE by RULE124] --->
                 (NOT (EXISTS C1 (CARDS-IN-HAND ME) (IN-SUIT C1 S0)))

 8:2             --- [by RULE159] --->
                 (EMPTY (SET-OF C1 (CARDS-IN-HAND ME) (IN-SUIT C1 S0)))

         (ACHIEVE (EMPTY (SET-OF C1 (CARDS-IN-HAND ME) (IN-SUIT C1 S0))))
 8:3     --- [REDUCE by RULE6] --->
         (UNTIL (VOID ME S0)
                (ACHIEVE (REMOVE-1-FROM (SET-OF C1 (CARDS-IN-HAND ME) (IN-SUIT C1 S0)))))

                       (REMOVE-1-FROM (SET-OF C1 (CARDS-IN-HAND ME) (IN-SUIT C1 S0)))
 8:4                   --- [REDUCE by RULE7] --->
                       (UNDO (AND [IN C1 (CARDS-IN-HAND ME)] [IN-SUIT C1 S0]))

 8:5                   --- [REDUCE by RULE35] --->
                       (AND [UNDO (IN C1 (CARDS-IN-HAND ME))] [IN-SUIT C1 S0])

                                     (CARDS-IN-HAND ME)
 8:6                                 --- [ELABORATE by RULE124] --->
                                     (SET-OF C2 (CARDS) (HAS ME C2))

                                  (IN C1 (SET-OF C2 (CARDS) (HAS ME C2)))
 8:7                              --- [REMOVE-QUANT by RULE10] --->
                                  (AND [IN C1 (CARDS)] [HAS ME C1])

                            (UNDO (AND [IN C1 (CARDS)] [HAS ME C1]))
 8:8                        --- [REDUCE by RULE35] --->
```

```
                               (AND [UNDO (HAS ME C1)] [IN C1 (CARDS)])

                                          (HAS ME C1)
8:9                                       --- [ELABORATE by RULE124] --->
                                          (AT C1 (HAND ME))

                                    (UNDO (AT C1 (HAND ME)))
8:10                                --- [RECOGNIZE by RULE368] --->
                                    (MOVE C1 (HAND ME) LOC1)

8:11                                --- [RECOGNIZE by RULE123] --->
                                    (PLAY ME C1)

                                    (IN C1 (CARDS))
8:12                                --- [EVAL by RULE346] --->
                                    T

                               (AND [PLAY ME C1] T)
8:13                           --- [SIMPLIFY by RULE343] --->
                               (PLAY ME C1)

                          (AND [PLAY ME C1] [IN-SUIT C1 S0])
8:14                      --- [RECOGNIZE by RULE123] --->
                          (PLAY-SUIT ME S0)

          [Final expression:]
          (UNTIL (VOID ME S0) (ACHIEVE (PLAY-SUIT ME S0)))

(ASSUMING (IN C1 (CARDS)))

<42>recordfile

RECORD FILE DSK: (DV8D07 . PZG) CLOSED 07-DEC-80 14:33:59
```

D.9. DERIV9: Decide if an Opponent is Void (Based on Distribution)

```
RECORD FILE DSK: (DV9D07 . PZG) OPENED 07-DEC-80 18:32:46
NIL
<21>p-links nil

(DERIV9 STARTED "24-APR-80 01:42:24" --DOMAIN: HEARTS --PROBLEM:
        (EVAL (VOID P0 S0)))


          [Initial expression:]
          (EVAL (VOID P0 S0))

                (VOID P0 S0)
9:1             --- [ELABORATE by RULE124] --->
                (NOT (EXISTS C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0)))

                     (EXISTS C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0))
> 9:2                --- [by RULE189] --->
               (SHOW (* (EXISTS C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0))
                        (H1 (DISJOINT (CARDS-IN-HAND P0) S1))))

                            (DISJOINT (CARDS-IN-HAND P0) S1)
| 9:3                       --- [ELABORATE by RULE124] --->
                            (NOT (OVERLAP (CARDS-IN-HAND P0) S1))

                                 (OVERLAP (CARDS-IN-HAND P0) S1)
| 9:4                            --- [ELABORATE by RULE124] --->
                                 (EXISTS X1 (CARDS-IN-HAND P0) (IN X1 S1))

                     (* (EXISTS C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0))
                        (H1 (NOT (EXISTS X1 (CARDS-IN-HAND P0) (IN X1 S1)))))
|> 9:5               --- [REDUCE by RULE212] --->
               (SHOW (* (EXISTS C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0))
                        (EXISTS X1 (CARDS-IN-HAND P0) (IN X1 S1))))

                     (* (EXISTS C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0))
                        (EXISTS X1 (CARDS-IN-HAND P0) (IN X1 S1)))
|| 9:6               --- [REDUCE by RULE192] --->
                     (* (IN-SUIT C1 S0) (IN C1 S1))
```

```
|| 9:7           --- [by RULE193] --->
                (= S1 (SET-OF C1 (DOMAIN C1 (IN-SUIT C1 S0)) (IN-SUIT C1 S0)))

                              (IN-SUIT C1 S0)
|| 9:8                        --- [ELABORATE by RULE124] --->
                              (= (SUIT-OF C1) S0)

                        (DOMAIN C1 (= (SUIT-OF C1) S0))
|| 9:9                  --- [by RULE195] --->
                        (DOMAIN C1 (SUIT-OF C1))

|| 9:10                 --- [by RULE196] --->
                        (DOMAIN SUIT-OF)

|| 9:11                 --- [FACT by RULE197] --->
                        (CARDS)

                  (SET-OF C1 (CARDS) (IN-SUIT C1 S0))
|| 9:12           --- [RECOGNIZE by RULE123] --->
                  (CARDS-IN-SUIT S0)

                (= S1 (CARDS-IN-SUIT S0))
|| 9:13         --- [REDUCE by RULE194] --->
                T
(S1 <- BINDING : (CARDS-IN-SUIT S0))

            (SHOW T)
|< 9:14     --- [SUCCEED by RULE34] --->
            (SHOW (= H1 (INVERSE (LAMBDA (EXISTS1) (NOT EXISTS1)))))

                         (LAMBDA (EXISTS1) (NOT EXISTS1))
| 9:15                   --- [SIMPLIFY by RULE215] --->
                         NOT

                (= H1 (INVERSE NOT))
| 9:16          --- [REDUCE by RULE194] --->
                T

            (SHOW T)
< 9:17      --- [SUCCEED by RULE190] --->
        (EVAL (NOT (H1 (DISJOINT (CARDS-IN-HAND P0) S1))))

                   H1
 9:18              --- [EVAL by RULE127] --->
                 (INVERSE NOT)

            (NOT ((INVERSE NOT) (DISJOINT (CARDS-IN-HAND P0) S1)))
 9:19       --- [SIMPLIFY by RULE216] --->
            (DISJOINT (CARDS-IN-HAND P0) S1)

                    S1
 9:20               --- [EVAL by RULE127] --->
                    (CARDS-IN-SUIT S0)

            (DISJOINT (CARDS-IN-HAND P0) (CARDS-IN-SUIT S0))
 9:21       --- [by RULE198] --->
            (PR-DISJOINT (CARDS-IN-HAND P0)
                         (CARDS-IN-SUIT S0)
                         (COMMON-SUPERSET (CARDS-IN-HAND P0) (CARDS-IN-SUIT S0)))

                         (COMMON-SUPERSET (CARDS-IN-HAND P0) (CARDS-IN-SUIT S0))
> 9:22                   --- [by RULE268] --->
               (CONSIDER (COMMON-SUPERSET (CARDS-IN-HAND P0) (CARDS-IN-SUIT S0)))

                                    (CARDS-IN-HAND P0)
| 9:23                              --- [ELABORATE by RULE124] --->
                                    (SET-OF C2 (CARDS) (HAS P0 C2))

                                    (CARDS-IN-SUIT S0)
| 9:24                              --- [ELABORATE by RULE124] --->
                                    (SET-OF C3 (CARDS) (IN-SUIT C3 S0))

                       (COMMON-SUPERSET (SET-OF C2 (CARDS) (HAS P0 C2))
                        (SET-OF C3 (CARDS) (IN-SUIT C3 S0)))
|. 9:26                --- [REDUCE by RULE199] --->
                       (CARDS)

               (CONSIDER (CARDS))
< 9:26         --- [by RULE269] --->
        (EVAL (PR-DISJOINT (CARDS-IN-HAND P0) (CARDS-IN-SUIT S0) (CARDS)))
```

```
               (PR-DISJOINT (CARDS-IN-HAND P0) (CARDS-IN-SUIT S0) (CARDS))
> 9:27         --- [by RULE200] --->
          (SHOW (= (CARDS-IN-HAND P0) (FILTER (CARDS-IN-HAND P0) OUT)))

               (= (CARDS-IN-HAND P0) (FILTER (CARDS-IN-HAND P0) OUT))
| 9:28         --- [by RULE218] --->
               (=> (IN X2 (CARDS-IN-HAND P0)) (OUT X2))

                       (CARDS-IN-HAND P0)
| 9:29                 --- [ELABORATE by RULE124] --->
                       (SET-OF C4 (CARDS) (HAS P0 C4))

                   (IN X2 (SET-OF C4 (CARDS) (HAS P0 C4)))
| 9:30             --- [REMOVE-QUANT by RULE369] --->
                   (HAS P0 X2)

                   (OUT X2)
| 9:31             --- [ELABORATE by RULE124] --->
                   (EXISTS P1 (OPPONENTS ME) (HAS P1 X2))

               (=> (HAS P0 X2) (EXISTS P1 (OPPONENTS ME) (HAS P1 X2)))
| 9:32         --- [by RULE329] --->
               (EXISTS P1 (OPPONENTS ME) (=> (HAS P0 X2) (HAS P1 X2)))

                       (=> (HAS P0 X2) (HAS P1 X2))
| 9:33                 --- [DISTRIBUTE by RULE43] --->
                       (AND [= P0 P1] [= X2 X2])

                            (= X2 X2)
| 9:34                      --- [EVAL by RULE179] --->
                            T

                       (AND [= P0 P1] T)
| 9:35                 --- [SIMPLIFY by RULE343] --->
                       (= P0 P1)

               (EXISTS P1 (OPPONENTS ME) (= P0 P1))
| 9:36         --- [REMOVE-QUANT by RULE162] --->
               (IN P0 (OPPONENTS ME))

          (SHOW (IN P0 (OPPONENTS ME)))
< 9:37    --- [ASSUME by RULE32] --->
          (EVAL (PR-DISJOINT (CARDS-IN-HAND P0)
                             (FILTER (CARDS-IN-SUIT S0) OUT)
                             (FILTER (CARDS) OUT)))

                             (FILTER (CARDS-IN-SUIT S0) OUT)
  9:38                       --- [RECOGNIZE by RULE123] --->
                             (CARDS-OUT-IN-SUIT S0)

                             (FILTER (CARDS) OUT)
  9:39                       --- [RECOGNIZE by RULE123] --->
                             (CARDS-OUT)

               (PR-DISJOINT (CARDS-IN-HAND P0)
                            (CARDS-OUT-IN-SUIT S0)
                            (CARDS-OUT))
  9:40         --- [ELABORATE by RULE124] --->
               (PR-DISJOINT-FORMULA (# (CARDS-IN-HAND P0))
                (# (CARDS-OUT-IN-SUIT S0))
                (# (CARDS-OUT)))

                                     (# (CARDS-IN-HAND P0))
  9:41                               --- [RECOGNIZE by RULE123] --->
                                     (#CARDS-IN-HAND P0)

                                     (# (CARDS-OUT-IN-SUIT S0))
  9:42                               --- [RECOGNIZE by RULE123] --->
                                     (#CARDS-OUT-IN-SUIT S0)

                                     (# (CARDS-OUT))
  9:43                               --- [RECOGNIZE by RULE123] --->
                                     (#CARDS-OUT)

          (EVAL (PR-DISJOINT-FORMULA (#CARDS-IN-HAND P0)
                 (#CARDS-OUT-IN-SUIT S0)
                 (#CARDS-OUT)))
  9:44    --- [by RULE202] --->
          (FUNCTION-OF S0
                   (DEPENDENCE (PR-DISJOINT-FORMULA (#CARDS-IN-HAND P0)
                                  (#CARDS-OUT-IN-SUIT S0)
                                  (#CARDS-OUT))
                                 S0))

                      (DEPENDENCE (PR-DISJOINT-FORMULA (#CARDS-IN-HAND P0)
                                  (#CARDS-OUT-IN-SUIT S0)
                                  (#CARDS-OUT))
                                 S0)
9:45                  --- [by RULE203] --->
                      (D+ (D* (DEPENDENCE (#CARDS-OUT-IN-SUIT S0) S0)
                              (DEPENDENCE (PR-DISJOINT-FORMULA #H #S #U) #S)))

9:46                  --- [SIMPLIFY by RULE178] --->
                      (D* (DEPENDENCE (#CARDS-OUT-IN-SUIT S0) S0)
                          (DEPENDENCE (PR-DISJOINT-FORMULA #H #S #U) #S))

         (FUNCTION-OF S0
                      (D* (DEPENDENCE (#CARDS-OUT-IN-SUIT S0) S0)
                          (DEPENDENCE (PR-DISJOINT-FORMULA #H #S #U) #S)))
9:47     --- [by RULE210] --->
         (FUNCTION-OF (#CARDS-OUT-IN-SUIT S0)
                      (D* INCREASING (DEPENDENCE (PR-DISJOINT-FORMULA #H #S #U) #S)))

                      (D* INCREASING (DEPENDENCE (PR-DISJOINT-FORMULA #H #S #U) #S))
9:48                  --- [SIMPLIFY by RULE341] --->
                      (DEPENDENCE (PR-DISJOINT-FORMULA #H #S #U) #S)

                                 (PR-DISJOINT-FORMULA #H #S #U)
9:49                             --- [ELABORATE by RULE124] --->
                                 (// (#CHOOSE (- #U #S) #H) (#CHOOSE #U #H))

                      (DEPENDENCE (// (#CHOOSE (- #U #S) #H) (#CHOOSE #U #H)) #S)
9:50                  --- [by RULE205] --->
                      (D+ (DEPENDENCE (#CHOOSE (- #U #S) #H) #S)
                          (D- (DEPENDENCE (#CHOOSE #U #H) #S)))

                              (DEPENDENCE (#CHOOSE #U #H) #S)
9:51                          --- [EVAL by RULE204] --->
                              NIL

                          (D- NIL)
9:52                      --- [COMPUTE by RULE236] --->
                          NIL

                      (D+ (DEPENDENCE (#CHOOSE (- #U #S) #H) #S) NIL)
9:53                  --- [SIMPLIFY by RULE343] --->
                      (DEPENDENCE (#CHOOSE (- #U #S) #H) #S)

9:54                  --- [REDUCE by RULE207] --->
                      (DEPENDENCE (- #U #S) #S)

9:55                  --- [by RULE205] --->
                      (D+ (DEPENDENCE #U #S) (D- (DEPENDENCE #S #S)))

                          (DEPENDENCE #U #S)
9:56                      --- [EVAL by RULE204] --->
                          NIL

                              (DEPENDENCE #S #S)
9:57                          --- [EVAL by RULE208] --->
                              INCREASING

                          (D- INCREASING)
9:58                      --- [COMPUTE by RULE236] --->
                          DECREASING

                      (D+ NIL DECREASING)
9:59                  --- [COMPUTE by RULE236] --->
                      DECREASING

         [Final expression:]
         (FUNCTION-OF (#CARDS-OUT-IN-SUIT S0) DECREASING)
NIL
<22>recordfile

RECORD FILE DSK: (DV9D07 . PZG) CLOSED 07-DEC-80 18:34:46
```

D.10. DERIV10: Count the Cards Out in a Suit

```
RECORD FILE DSK: (D10D07 . PZG) OPENED 07-DEC-80 18:36:24
NIL
<47>p-links nil

(DERIV10 STARTED "24-APR-80 20:04:56" --DOMAIN: HEARTS --PROBLEM:
        (EVAL (#CARDS-OUT-IN-SUIT S0)))


         [Initial expression:]
         (EVAL (#CARDS-OUT-IN-SUIT S0))

              (#CARDS-OUT-IN-SUIT S0)
 10:1         --- [ELABORATE by RULE124] --->
              (# (CARDS-OUT-IN-SUIT S0))

                 (CARDS-OUT-IN-SUIT S0)
 10:2            --- [ELABORATE by RULE124] --->
                 (FILTER (CARDS-IN-SUIT S0) OUT)

 10:3            --- [ELABORATE by RULE124] --->
                 (SET-OF X1 (CARDS-IN-SUIT S0) (OUT X1))

              (# (SET-OF X1 (CARDS-IN-SUIT S0) (OUT X1)))
> 10:4        --- [TRY-THIS by RULE370] --->
         (SHOW (NOT (DURING (ROUND-IN-PROGRESS) (CAUSE (OUT X1)))))

                                   (OUT X1)
| 10:5                             --- [ELABORATE by RULE124] --->
                                   (EXISTS P1 (OPPONENTS ME) (HAS P1 X1))

                            (CAUSE (EXISTS P1 (OPPONENTS ME) (HAS P1 X1)))
| 10:6                      --- [by RULE329] --->
                            (EXISTS P1 (OPPONENTS ME) (CAUSE (HAS P1 X1)))

                                           (HAS P1 X1)
| 10:7                                     --- [ELABORATE by RULE124] --->
                                           (AT X1 (HAND P1))

                                    (CAUSE (AT X1 (HAND P1)))
| 10:8                              --- [RECOGNIZE by RULE358] --->
                                    (MOVE X1 LOC1 (HAND P1))

| 10:9                              --- [RECOGNIZE by RULE123] --->
                                    (GET-CARD P1 X1 LOC1)

                   (DURING (ROUND-IN-PROGRESS)
                           (EXISTS P1 (OPPONENTS ME) (GET-CARD P1 X1 LOC1)))
| 10:10            --- [COMPUTE by RULE57] --->
                   NIL

              (NOT NIL)
| 10:11       --- [COMPUTE by RULE236] --->
              T

         (SHOW T)
< 10:12  --- [SUCCEED by RULE34] --->
         (EVAL (- (# (SET-OF X1 (CARDS-IN-SUIT S0)
                             (BEFORE (CURRENT ROUND-IN-PROGRESS) (OUT X1))))
                  (# (SET-OF X1 (CARDS-IN-SUIT S0)
                             (WAS-DURING (CURRENT ROUND-IN-PROGRESS) (UNDO (OUT X1)))))))

                             (BEFORE (CURRENT ROUND-IN-PROGRESS) (OUT X1))
> 10:13                      --- [by RULE268] --->
                   (CONSIDER (BEFORE (CURRENT ROUND-IN-PROGRESS) (OUT X1)))

                                     (OUT X1)
| 10:14                              --- [REDUCE by RULE301] --->
                                     (NOT (OR [IN-POT X1] [AT X1 HOLE] [HAS ME X1]))

                             (BEFORE (CURRENT ROUND-IN-PROGRESS)
                                     (NOT (OR [IN-POT X1] [AT X1 HOLE] [HAS ME X1])))
| 10:15                      --- [DISTRIBUTE by RULE371] --->
                             (NOT (BEFORE (CURRENT ROUND-IN-PROGRESS)
                                          (OR [IN-POT X1] [AT X1 HOLE] [HAS ME X1])))
                                     (BEFORE (CURRENT ROUND-IN-PROGRESS)
                                             (OR [IN-POT X1] [AT X1 HOLE] [HAS ME X1]))
| 10:16                              --- [DISTRIBUTE by RULE371] --->
                                     (OR [BEFORE (CURRENT ROUND-IN-PROGRESS) (IN-POT X1)]
                                         [BEFORE (CURRENT ROUND-IN-PROGRESS) (AT X1 HOLE)]
                                         [BEFORE (CURRENT ROUND-IN-PROGRESS) (HAS ME X1)])

                                         (BEFORE (CURRENT ROUND-IN-PROGRESS) (IN-POT X1))
| 10:17                                  --- [FACT by RULE372] --->
                                         NIL

                                     (OR NIL [BEFORE (CURRENT ROUND-IN-PROGRESS) (AT X1 HOLE)]
                                         [BEFORE (CURRENT ROUND-IN-PROGRESS) (HAS ME X1)])
| 10:18                              --- [SIMPLIFY by RULE176] --->
                                     (OR [BEFORE (CURRENT ROUND-IN-PROGRESS) (AT X1 HOLE)]
                                         [BEFORE (CURRENT ROUND-IN-PROGRESS) (HAS ME X1)])

                                                 (AT X1 HOLE)
| 10:19                                          --- [ASSUME by RULE373] --->
                                                 NIL

                                         (BEFORE (CURRENT ROUND-IN-PROGRESS) NIL)
| 10:20                                  --- [EVAL by RULE374] --->
                                         NIL

                                     (OR NIL [BEFORE (CURRENT ROUND-IN-PROGRESS) (HAS ME X1)])
| 10:21                              --- [SIMPLIFY by RULE341] --->
                                     (BEFORE (CURRENT ROUND-IN-PROGRESS) (HAS ME X1))

                       (CONSIDER (NOT (BEFORE (CURRENT ROUND-IN-PROGRESS) (HAS ME X1))))
< 10:22                --- [by RULE269] --->
             (EVAL (- (# (SET-OF X1 (CARDS-IN-SUIT S0)
                                    (NOT (BEFORE (CURRENT ROUND-IN-PROGRESS) (HAS ME X1)))))
                      (# (SET-OF X1 (CARDS-IN-SUIT S0)
                                    (WAS-DURING (CURRENT ROUND-IN-PROGRESS) (UNDO (OUT X1)))))))

                      (# (SET-OF X1 (CARDS-IN-SUIT S0)
                                    (NOT (BEFORE (CURRENT ROUND-IN-PROGRESS) (HAS ME X1)))))
  10:23               --- [by RULE375] --->
                      (- (# (CARDS-IN-SUIT S0))
                         (# (SET-OF X1 (CARDS-IN-SUIT S0)
                                       (BEFORE (CURRENT ROUND-IN-PROGRESS) (HAS ME X1)))))

                         (# (CARDS-IN-SUIT S0))
  10:24                  --- [FACT by RULE376] --->
                         13

                                                 (UNDO (OUT X1))
> 10:25                                          --- [by RULE268] --->
                                  (CONSIDER (UNDO (OUT X1)))

                                                      (OUT X1)
| 10:26                                               --- [ELABORATE by RULE124] --->
                                                      (EXISTS P3 (OPPONENTS ME) (HAS P3 X1))

                                                (UNDO (EXISTS P3 (OPPONENTS ME) (HAS P3 X1)))
| 10:27                                         --- [by RULE329] --->
                                                (EXISTS P3 (OPPONENTS ME) (UNDO (HAS P3 X1)))

                                                          (HAS P3 X1)
| 10:28                                                   --- [ELABORATE by RULE124] --->
                                                          (AT X1 (HAND P3))

                                                    (UNDO (AT X1 (HAND P3)))
| 10:29                                             --- [RECOGNIZE by RULE368] --->
                                                    (MOVE X1 (HAND P3) LOC2)

| 10:30                                             --- [RECOGNIZE by RULE123] --->
                                                    (PLAY P3 X1)

                                  (CONSIDER (EXISTS P3 (OPPONENTS ME) (PLAY P3 X1)))
< 10:31                           --- [by RULE269] --->
             (EVAL (- (- 13
                         (# (SET-OF X1 (CARDS-IN-SUIT S0)
                                       (BEFORE (CURRENT ROUND-IN-PROGRESS) (HAS ME X1)))))
                      (# (SET-OF X1 (CARDS-IN-SUIT S0)
                                    (WAS-DURING (CURRENT ROUND-IN-PROGRESS)
                                                (EXISTS P3 (OPPONENTS ME) (PLAY P3 X1)))))))

             [Final expression:]
             (EVAL (- (- 13
                      (# (SET-OF X1 (CARDS-IN-SUIT S0)
                                  (BEFORE (CURRENT ROUND-IN-PROGRESS) (HAS ME X1)))))
                   (# (SET-OF X1 (CARDS-IN-SUIT S0)
                              (WAS-DURING (CURRENT ROUND-IN-PROGRESS)
                                          (EXISTS P3 (OPPONENTS ME) (PLAY P3 X1)))))))
NIL
<48>recordfile

RECORD FILE DSK: (D10D07 . PZG) CLOSED 07-DEC-80 18:37:06
```

D.11. DERIV11: Decide if an Opponent is Void (Based on Behavior)

```
RECORD FILE DSK: (D11D07 . PZG) OPENED 07-DEC-80 18:37:28
NIL
<82>p-links nil

(DERIV11 STARTED "24-APR-80 22:47:57" --DOMAIN: HEARTS --PROBLEM:
         (EVAL (VOID P0 S0)))


          [Initial expression:]
          (EVAL (VOID P0 S0))

> 11:1    --- [by RULE227] --->
    (SHOW (LEGAL P1 C1))

          (LEGAL P1 C1)
|> 11:2   --- [FACT by RULE228] --->
    (SHOW (= C1 (CARD-OF P1)))

          (= C1 (CARD-OF P1))
|| 11:3   --- [REDUCE by RULE194] --->
          T
(C1 <- BINDING : (CARD-OF P1))

    (SHOW T)
|< 11:4
    --- [SUCCEED by RULE53] --->
    (SHOW T)

< 11:5
    --- [SUCCEED by RULE34] --->
          (EVAL (=> (LEGAL P1 C1) (VOID P0 S0)))

                            C1
 11:6                       --- [EVAL by RULE127] --->
                            (CARD-OF P1)

                     (LEGAL P1 (CARD-OF P1))
 11:7                --- [ELABORATE by RULE124] --->
                     (AND [HAS P1 (CARD-OF P1)]
                          [=> (LEADING P1)
                              (OR [CAN-LEAD-HEARTS P1] [NEQ (SUIT-OF (CARD-OF P1)) H])]
                          [=> (FOLLOWING P1)
                              (OR [VOID P1 (SUIT-LED)] [IN-SUIT (CARD-OF P1) (SUIT-LED)])])

                 (=> (AND [HAS P1 (CARD-OF P1)]
                          [=> (LEADING P1)
                              (OR [CAN-LEAD-HEARTS P1] [NEQ (SUIT-OF (CARD-OF P1)) H])]
                          [=> (FOLLOWING P1)
                              (OR [VOID P1 (SUIT-LED)] [IN-SUIT (CARD-OF P1) (SUIT-LED)])])
                     (VOID P0 S0))
 11:8            --- [REDUCE by RULE224] --->
                 (=> (=> (FOLLOWING P1)
                         (OR [VOID P1 (SUIT-LED)] [IN-SUIT (CARD-OF P1) (SUIT-LED)]))
                     (VOID P0 S0))

 11:9            --- [REDUCE by RULE226] --->
                 (AND [FOLLOWING P1]
                      [=> (OR [VOID P1 (SUIT-LED)] [IN-SUIT (CARD-OF P1) (SUIT-LED)])
                          (VOID P0 S0)])

                      (=> (OR [VOID P1 (SUIT-LED)] [IN-SUIT (CARD-OF P1) (SUIT-LED)])
                          (VOID P0 S0))
 11:10               --- [REDUCE by RULE226] --->
                     (AND [NOT (OR [IN-SUIT (CARD-OF P1) (SUIT-LED)])]
                          [=> (VOID P1 (SUIT-LED)) (VOID P0 S0)])

                          (=> (VOID P1 (SUIT-LED)) (VOID P0 S0))
 11:11                    --- [DISTRIBUTE by RULE43] --->
                          (AND [= P1 P0] [= (SUIT-LED) S0])

                               (= P1 P0)
 11:12                         --- [REDUCE by RULE194] --->
                               T
(P1 <- BINDING : P0)

                          (AND T [= (SUIT-LED) S0])
 11:13                    --- [SIMPLIFY by RULE341] --->
                          (= (SUIT-LED) S0)

                               (OR [IN-SUIT (CARD-OF P1) (SUIT-LED)])
 11:14                         --- [SIMPLIFY by RULE178] --->
                               (IN-SUIT (CARD-OF P1) (SUIT-LED))

                                P1
 11:15                          --- [EVAL by RULE127] --->
                                P0

                                                P1
 11:16                                          --- [EVAL by RULE127] --->
                                                P0

                (AND [FOLLOWING P0]
                     [AND [NOT (IN-SUIT (CARD-OF P0) (SUIT-LED))] [= (SUIT-LED) S0]])
 11:17          --- [SIMPLIFY by RULE177] --->
                (AND [NOT (IN-SUIT (CARD-OF P0) (SUIT-LED))]
                     [= (SUIT-LED) S0]
                     [FOLLOWING P0])

                     (NOT (IN-SUIT (CARD-OF P0) (SUIT-LED)))
 11:18               --- [RECOGNIZE by RULE123] --->
                     (BREAKING-SUIT P0)

         [Final expression:]
         (EVAL (AND [BREAKING-SUIT P0] [= (SUIT-LED) S0] [FOLLOWING P0]))
NIL
<33>recordfile

RECORD FILE DSK: (D11D07 . PZG) CLOSED 07-DEC-80 16:38:34
```

D.12. DERIV12: Decide if an Opponent is Void (Based on Past)

```
RECORD FILE DSK: (D12D07 . PZG) OPENED 07-DEC-80 16:04:36
NIL
<37>p-links nil

(DERIV12 STARTED "07-DEC-80 14:59:15" #LINKS 13 --DOMAIN: HEARTS
         --PROBLEM: (EVAL (VOID P0 S0)))


         [Initial expression:]
         (EVAL (VOID P0 S0))

              VOID P0 S0)
> 12:1        --- [SUFFICE by RULE234] --->
         (SHOW (NOT (DURING (ROUND-IN-PROGRESS) (UNDO (VOID P0 S0)))))

                                  (VOID P0 S0)
| 12:2                            --- [ELABORATE by RULE124] --->
                                  (NOT (EXISTS C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0)))

                           (UNDO (NOT (EXISTS C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0))))
| 12:3                     --- [RECOGNIZE by RULE377] --->
                           (CAUSE (EXISTS C1 (CARDS-IN-HAND P0) (IN-SUIT C1 S0)))

                                          (CARDS-IN-HAND P0)
| 12:4                                    --- [ELABORATE by RULE124] --->
                                          (SET-OF C2 (CARDS) (HAS P0 C2))
```

```
                                   (EXISTS C1 (SET-OF C2 (CARDS) (HAS P0 C2)) (IN-SUIT C1 S0))
| 12:5                             --- [by RULE135] --->
                                   (EXISTS C1 (CARDS) (AND [HAS P0 C1] [IN-SUIT C1 S0]))

                              (CAUSE (EXISTS C1 (CARDS) (AND [HAS P0 C1] [IN-SUIT C1 S0])))
| 12:6                        --- [by RULE329] --->
                              (EXISTS C1 (CARDS) (CAUSE (AND [HAS P0 C1] [IN-SUIT C1 S0])))

                                   (CAUSE (AND [HAS P0 C1] [IN-SUIT C1 S0]))
| 12:7                             --- [REDUCE by RULE35] --->
                                   (AND [CAUSE (HAS P0 C1)] [IN-SUIT C1 S0])

                                            (HAS P0 C1)
| 12:8                                      --- [ELABORATE by RULE124] --->
                                            (AT C1 (HAND P0))

                                       (CAUSE (AT C1 (HAND P0)))
| 12:9                                 --- [RECOGNIZE by RULE358] --->
                                       (MOVE C1 LOC1 (HAND P0))

| 12:10                                --- [RECOGNIZE by RULE123] --->
                                       (GET-CARD P0 C1 LOC1)

                        (DURING (ROUND-IN-PROGRESS)
                              (EXISTS C1 (CARDS) (AND [GET-CARD P0 C1 LOC1] [IN-SUIT C1 S0])))
| 12:11                 --- [COMPUTE by RULE57] --->
                        NIL

                    (NOT NIL)
| 12:12             --- [COMPUTE by RULE236] --->
                    T

               (SHOW T)
< 12:13        --- [SUCCEED by RULE34] --->
               (EVAL (WAS-DURING (CURRENT ROUND-IN-PROGRESS) (VOID P0 S0)))

               [Final expression:]
               (EVAL (WAS-DURING (CURRENT ROUND-IN-PROGRESS) (VOID P0 S0)))
<38>recordfile

RECORD FILE DSK: (D12D07 . PZG) CLOSED 07-DEC-80 15:05:26
```

D.13. DERIV13: Heuristic Search to Generate Cantus

```
RECORD FILE DSK: (D13D07 . PZG) OPENED 07-DEC-80 18:40:00
NIL
<6>p-links nil

(DERIV13 STARTED "25-APR-80 18:37:30" #LINKS 116 --DOMAIN: MUSIC
       --PROBLEM: (ACHIEVE (LEGAL-CANTUS! (CANTUS))))


       [Initial expression:]
       (ACHIEVE (LEGAL-CANTUS! (CANTUS)))

                      (LEGAL-CANTUS! (CANTUS))
 13:1                 --- [by RULE306] --->
                      HSM1
(HSM1 <- PROBLEM : (LEGAL-CANTUS! (CANTUS)))
(HSM1 <- OBJECT : (CANTUS))
(HSM1 <- CHOICE-SEQ : (CHOICE-SEQ-OF (CANTUS)))

> 13:2                --- [by RULE312] --->
    (CONSIDER-PROP (CHOICE-SEQ-OF (CANTUS)))

                                 (CANTUS)
  13:3                           --- [ELABORATE by RULE124] --->
                                 (EACH I1 (LB:UB 1 (CANTUS-LENGTH)) (CHOOSE-NOTE I1))

                      (CHOICE-SEQ-OF
                       (EACH I1 (LB:UB 1 (CANTUS-LENGTH)) (CHOOSE-NOTE I1)))
   1 4                --- [DISTRIBUTE by RULE307] --->
                      (APPLY APPEND
                          (EACH I1 (LB:UB 1 (CANTUS-LENGTH))
                                (CHOICE-SEQ-OF (CHOOSE-NOTE I1))))

                                              (CHOOSE-NOTE I1)
| 13:5                                        --- [ELABORATE by RULE124] --->
                                              (CHOOSE (NOTE I1) (TONES) (NOTE I1))

                                (CHOICE-SEQ-OF (CHOOSE (NOTE I1) (TONES) (NOTE I1)))
| 13:6                          --- [RECOGNIZE by RULE309] --->
                                (LIST (CHOOSE (NOTE I1) (TONES) (NOTE I1)))

                    (APPLY APPEND
                          (EACH I1 (LB:UB 1 (CANTUS-LENGTH))
                                (LIST (CHOOSE (NOTE I1) (TONES) (NOTE I1)))))
| 13:7              --- [SIMPLIFY by RULE310] --->
                    (EACH I1 (LB:UB 1 (CANTUS-LENGTH))
                          (CHOOSE (NOTE I1) (TONES) (NOTE I1)))

   (CONSIDER-PROP
    (EACH I1 (LB:UB 1 (CANTUS-LENGTH))
          (CHOOSE (NOTE I1) (TONES) (NOTE I1))))
< 13:8
   --- [SUCCEED by RULE313] --->
         (ACHIEVE HSM1)
(HSM1 <- CHOICE-SEQ :
      (EACH I1 (LB:UB 1 (CANTUS-LENGTH))
            (CHOOSE (NOTE I1) (TONES) (NOTE I1))))

                    HSM1
 13:9               --- [ELABORATE by RULE311] --->
                    HSM1
(HSM1 <- SEQUENCE : NOTE)
(HSM1 <- CHOICES : (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))))
(HSM1 <- CHOICE-SETS : (LAMBDA (I1) (TONES)))
(HSM1 <- INDICES : (LB:UB 1 (CANTUS-LENGTH)))
(HSM1 <- INDEX : I1)
(HSM1 <- VARIABLES : NIL)
(HSM1 <- BINDINGS : NIL)
(HSM1 <- INITIAL-PATH : NIL)
(HSM1 <- COMPLETION-TEST :
      (LAMBDA (PATH) (= (# PATH) (# (LB:UB 1 (CANTUS-LENGTH))))))

> 13:10             --- [by RULE314] --->
   (CONSIDER-PROP
    (REFORMULATE (LEGAL-CANTUS! (CANTUS))
                 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))))

                                              (CANTUS)
| 13:11                                       --- [ELABORATE by RULE124] --->
                                              (EACH I2 (LB:UB 1 (CANTUS-LENGTH)) (CHOOSE-NOTE I2))

                                                    (CHOOSE-NOTE I2)
| 13:12                                             --- [ELABORATE by RULE124] --->
                                                    (CHOOSE (NOTE I2) (TONES) (NOTE I2))

| 13:13                                             --- [REDUCE by RULE238] --->
                                                    (NOTE I2)

                                              (EACH I2 (LB:UB 1 (CANTUS-LENGTH)) (NOTE I2))
| 13:14                                       --- [RECOGNIZE by RULE123] --->
                                              (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))

                    (REFORMULATE (LEGAL-CANTUS! (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))))
                                (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))))
| 13:15             --- [by RULE315] --->
                    ((LAMBDA (PROJECT1) (LEGAL-CANTUS! PROJECT1))
                     (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))))

                            (LEGAL-CANTUS! PROJECT1)
| 13:16                     --- [ELABORATE by RULE124] --->
                            (AND [IN (# PROJECT1) (LB:UB 8 16)]
                                 [FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                                        (USABLE-INTERVAL! INTERVAL1)]
                                 [=< (MELODIC-RANGE PROJECT1) (MAJOR 10)]
                                 [= (#OCCURRENCES (CLIMAX PROJECT1) PROJECT1) 1])

   (CONSIDER-PROP
    ((LAMBDA (PROJECT1)
      (AND [IN (# PROJECT1) (LB:UB 8 16)]
           [FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                  (USABLE-INTERVAL! INTERVAL1)]
           [=< (MELODIC-RANGE PROJECT1) (MAJOR 10)]
           [= (#OCCURRENCES (CLIMAX PROJECT1) PROJECT1) 1]))
     (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))))
< 13:17
   --- [SUCCEED by RULE313] --->
         (ACHIEVE HSM1)
(HSM1 <- REFORMULATED-PROBLEM :
     ((LAMBDA (PROJECT1)
       (AND [IN (# PROJECT1) (LB:UB 8 16)]
            [FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                    (USABLE-INTERVAL! INTERVAL1)]
            [=< (MELODIC-RANGE PROJECT1) (MAJOR 10)]
            [= (#OCCURRENCES (CLIMAX PROJECT1) PROJECT1) 1]))
      (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))))

                     HSM1
 13:18               --- [by RULE317] --->
                     HSM1
(HSM1 <- PATH : PROJECT1)
(HSM1 <- STEP-ORDER : NIL)
(HSM1 <- STEP-TEST : T)
(HSM1 <- PATH-ORDER : NIL)
(HSM1 <- PATH-TEST : T)
(HSM1 <- SOLUTION-TEST :
     (LAMBDA (PROJECT1)
      (AND [IN (# PROJECT1) (LB:UB 8 16)]
           [FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                   (USABLE-INTERVAL! INTERVAL1)]
           [=< (MELODIC-RANGE PROJECT1) (MAJOR 10)]
           [= (#OCCURRENCES (CLIMAX PROJECT1) PROJECT1) 1])))

 13:19               --- [by RULE378] --->
                     HSM1
(HSM1 <- COMPLETION-TEST :
     (LAMBDA (PROJECT1) (IN (# PROJECT1) (LB:UB 8 16))))
(HSM1 <- SOLUTION-TEST :
     (LAMBDA (PROJECT1)
      (AND [FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                   (USABLE-INTERVAL! INTERVAL1)]
           [=< (MELODIC-RANGE PROJECT1) (MAJOR 10)]
           [= (#OCCURRENCES (CLIMAX PROJECT1) PROJECT1) 1])))

> 13:20              --- [by RULE323] --->
   (CONSIDER-PROP
    (AND T
         [LAMBDA (PROJECT1)
          (=> (=> (FORALL INTERVAL1 (INTERVALS-OF (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))))
                          (USABLE-INTERVAL! INTERVAL1))
                  (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                          (USABLE-INTERVAL! INTERVAL1)))
              (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                      (USABLE-INTERVAL! INTERVAL1)))]))

                     (AND T
                          [LAMBDA (PROJECT1)
                           (=> (=> (FORALL INTERVAL1 (INTERVALS-OF (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))))
                                           (USABLE-INTERVAL! INTERVAL1))
                                   (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                                           (USABLE-INTERVAL! INTERVAL1)))
                               (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                                       (USABLE-INTERVAL! INTERVAL1)))])
| 13:21              --- [SIMPLIFY by RULE341] --->
                     (LAMBDA (PROJECT1)
                      (=> (=> (FORALL INTERVAL1 (INTERVALS-OF (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))))
                                      (USABLE-INTERVAL! INTERVAL1))
                              (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                                      (USABLE-INTERVAL! INTERVAL1)))
                          (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                                  (USABLE-INTERVAL! INTERVAL1))))

                               (=> (FORALL INTERVAL1 (INTERVALS-OF (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))))
                                           (USABLE-INTERVAL! INTERVAL1))
                                   (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                                           (USABLE-INTERVAL! INTERVAL1)))
|> 13:22                       --- [REDUCE by RULE379] --->
                            (SHOW (SUBSET (INTERVALS-OF PROJECT1)
                                          (INTERVALS-OF (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))))))

                               (SUBSET (INTERVALS-OF PROJECT1)
                                       (INTERVALS-OF (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))))
||> 13:23                      --- [REDUCE by RULE30] --->
```

```
                         (SHOW (SUBSET PROJECT1 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))))

||< 13:24                --- [ASSUME by RULE221] --->
                         (SHOW T)
((SUBSET PROJECT1 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))))
 <- ;
 (-> T IF
     (= (SUBSET PROJECT1 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))) T)))

|< 13:25                 --- [SUCCEED by RULE53] --->
    (CONSIDER-PROP
     (LAMBDA (PROJECT1)
      (=> T
          (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                  (USABLE-INTERVAL! INTERVAL1)))))

                           (=> T
                               (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                                       (USABLE-INTERVAL! INTERVAL1)))
| 13:26                    --- [SIMPLIFY by RULE12] --->
                           (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                                   (USABLE-INTERVAL! INTERVAL1))

    (CONSIDER-PROP
     (LAMBDA (PROJECT1)
      (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
              (USABLE-INTERVAL! INTERVAL1))))
< 13:27
    --- [SUCCEED by RULE313] --->
          (ACHIEVE HSM1)
(HSM1 <- PATH-TEST :
      (LAMBDA (PROJECT1)
       (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
               (USABLE-INTERVAL! INTERVAL1))))

                   HSM1
> 13:28            --- [by RULE327] --->
             (SHOW (= (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                              (USABLE-INTERVAL! INTERVAL1))
                      (FORALL I1 (INDICES-OF PROJECT1) Q1)))

                              (INTERVALS-OF PROJECT1)
| 13:29                       --- [ELABORATE by RULE124] --->
                              (DISTRIBUTE I4 (LB:UB 2 (# PROJECT1)),
                                          (INTERVAL (PROJECT1 (- I4 1)) (PROJECT1 I4)))

                      (FORALL INTERVAL1
                              (DISTRIBUTE I4 (LB:UB 2 (# PROJECT1))
                                          (INTERVAL (PROJECT1 (- I4 1)) (PROJECT1 I4)))
                              (USABLE-INTERVAL! INTERVAL1))
| 13:30               --- [REMOVE-QUANT by RULE13] --->
                      (FORALL I4 (LB:UB 2 (# PROJECT1))
                              (USABLE-INTERVAL! (INTERVAL (PROJECT1 (- I4 1)) (PROJECT1 I4))))

                   (= (FORALL I4 (LB:UB 2 (# PROJECT1))
                              (USABLE-INTERVAL! (INTERVAL (PROJECT1 (- I4 1)) (PROJECT1 I4))))
                     (FORALL I1 (INDICES-OF PROJECT1) Q1))
| 13:31            --- [REDUCE by RULE380] --->
                   (= Q1
                     (OR [USABLE-INTERVAL! (INTERVAL (PROJECT1 (- I1 1)) (PROJECT1 I1))]
                          [IN I1 (DIFF (INDICES-OF PROJECT1) (LB:UB 2 (# PROJECT1)))]))

                                    (INDICES-OF PROJECT1)
| 13:32                             --- [ELABORATE by RULE381] --->
                                    (LB:UB 1 (# PROJECT1))

                              (DIFF (LB:UB 1 (# PROJECT1)) (LB:UB 2 (# PROJECT1)))
| 13:33                       --- [REDUCE by RULE382] --->
                              (LB:UB 1 (- 2 1))

                                     (- 2 1)
| 13:34                              --- [COMPUTE by RULE236] --->
                                     1

                              (LB:UB 1 1)
| 13:35                       --- [SIMPLIFY by RULE253] --->
                              (SET 1)

                          (IN I1 (SET 1))
| 13:36                   --- [DISTRIBUTE by RULE182] --->
                          (OR [= I1 1])
```

```
| 13:37                 --- [SIMPLIFY by RULE178] --->
                        (= I1 1)

                  (= Q1
                     (OR [USABLE-INTERVAL! (INTERVAL (PROJECT1 (- I1 1)) (PROJECT1 I1))]
                         [= I1 1]))
| 13:38           --- [REDUCE by RULE194] --->
                  T
(Q1 <- BINDING :
    (OR [USABLE-INTERVAL! (INTERVAL (PROJECT1 (- I1 1)) (PROJECT1 I1))]
        [= I1 1]))

            (SHOW T)
< 13:39     --- [SUCCEED by RULE34] --->
         (ACHIEVE (THEN $CONSIDER-PROP HSM1 STEP-TEST
                        (AND T [LAMBDA (I1 NOTE1) (THEN $SUBST NOTE1 (PROJECT1 I1) Q1)])))

                        (AND T [LAMBDA (I1 NOTE1) (THEN $SUBST NOTE1 (PROJECT1 I1) Q1)])
 13:40                  --- [SIMPLIFY by RULE341] --->
                        (LAMBDA (I1 NOTE1) (THEN $SUBST NOTE1 (PROJECT1 I1) Q1))

                                     Q1
 13:41                               --- [EVAL by RULE127] --->
                                     (OR [USABLE-INTERVAL! (INTERVAL (PROJECT1 (- I1 1)) (PROJECT1 I1))]
                                         [= I1 1])

                              (THEN $SUBST NOTE1 (PROJECT1 I1)
                                     (OR [USABLE-INTERVAL! (INTERVAL (PROJECT1 (- I1 1)) (PROJECT1 I1))]
                                         [= I1 1]))
 13:42                        --- [SIMPLIFY by RULE331] --->
                              (OR [USABLE-INTERVAL! (INTERVAL (PROJECT1 (- I1 1)) NOTE1)]
                                  [= I1 1])

                  (THEN $CONSIDER-PROP HSM1 STEP-TEST
                        (LAMBDA (I1 NOTE1)
                         (OR [USABLE-INTERVAL! (INTERVAL (PROJECT1 (- I1 1)) NOTE1)]
                             [= I1 1])))
 13:43            --- [SIMPLIFY by RULE334] --->
                  HSM1
(HSM1 <- STEP-TEST :
      (LAMBDA (I1 NOTE1)
       (OR [USABLE-INTERVAL! (INTERVAL (PROJECT1 (- I1 1)) NOTE1)]
           [= I1 1])))

> 13:44             --- [by RULE323] --->
    (CONSIDER-PROP
     (AND [LAMBDA (PROJECT1)
           (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                   (USABLE-INTERVAL! INTERVAL1))]
          [LAMBDA (PROJECT1)
           (=> (=> (=< (MELODIC-RANGE (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))))
                       (MAJOR 10))
                   (=< (MELODIC-RANGE PROJECT1) (MAJOR 10)))
               (=< (MELODIC-RANGE PROJECT1) (MAJOR 10)))]))

                               (=> (=< (MELODIC-RANGE (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))))
                                       (MAJOR 10))
                                   (=< (MELODIC-RANGE PROJECT1) (MAJOR 10)))
|> 13:45                       --- [REDUCE by RULE383] --->
                           (SHOW (=< (MELODIC-RANGE PROJECT1)
                                     (MELODIC-RANGE (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))))))

                               (=< (MELODIC-RANGE PROJECT1)
                                   (MELODIC-RANGE (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))))
||> 13:46                      --- [REDUCE by RULE30] --->
                           (SHOW (SUBSET PROJECT1 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))))

                               (SUBSET PROJECT1 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))))
||| 13:47                      --- [EVAL by RULE346] --->
                               T

                           (SHOW T)
||< 13:48                  --- [SUCCEED by RULE63] --->
                           (SHOW T)

|< 13:49                   --- [SUCCEED by RULE63] --->
    (CONSIDER-PROP
     (AND [LAMBDA (PROJECT1)
           (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                   (USABLE-INTERVAL! INTERVAL1))]
          [LAMBDA (PROJECT1) (=> T (=< (MELODIC-RANGE PROJECT1) (MAJOR 10)))]))

                                (=> T (=< (MELODIC-RANGE PROJECT1) (MAJOR 10)))
| 13:50                         --- [SIMPLIFY by RULE12] --->
                                (=< (MELODIC-RANGE PROJECT1) (MAJOR 10))

    (CONSIDER-PROP
     (AND [LAMBDA (PROJECT1)
           (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                   (USABLE-INTERVAL! INTERVAL1))]
          [LAMBDA (PROJECT1) (=< (MELODIC-RANGE PROJECT1) (MAJOR 10))]))
< 13:51
    --- [SUCCEED by RULE313] --->
          (ACHIEVE HSM1)
(HSM1 <- PATH-TEST :
      (AND [LAMBDA (PROJECT1)
            (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                    (USABLE-INTERVAL! INTERVAL1))]
           [LAMBDA (PROJECT1) (=< (MELODIC-RANGE PROJECT1) (MAJOR 10))]))

                   HSM1
 13:52             --- [REFINE by RULE384] --->
                   (THEN $CONSIDER-PROP HSM1 CHOICE-SETS
                         (LAMBDA (I1)
                          (SET-OF NOTE1 (TONES)
                                  (OR [USABLE-INTERVAL! (INTERVAL (PROJECT1 (- I1 1)) NOTE1)]
                                      [= I1 1]))))
(HSM1 <- STEP-TEST : T)

                                 (SET-OF NOTE1 (TONES)
                                         (OR [USABLE-INTERVAL! (INTERVAL (PROJECT1 (- I1 1)) NOTE1)]
                                             [= I1 1]))
 13:53                           --- [DISTRIBUTE by RULE385] --->
                                 (IF (= I1 1)
                                     (TONES)
                                     (SET-OF NOTE1 (TONES)
                                             (OR [USABLE-INTERVAL! (INTERVAL (PROJECT1 (- I1 1)) NOTE1)])))

                                             (OR [USABLE-INTERVAL! (INTERVAL (PROJECT1 (- I1 1)) NOTE1)])
 13:54                                       --- [SIMPLIFY by RULE178] --->
                                             (USABLE-INTERVAL! (INTERVAL (PROJECT1 (- I1 1)) NOTE1))

                   (THEN $CONSIDER-PROP HSM1 CHOICE-SETS
                         (LAMBDA (I1)
                          (IF (= I1 1)
                              (TONES)
                              (SET-OF NOTE1 (TONES)
                                      (USABLE-INTERVAL! (INTERVAL (PROJECT1 (- I1 1)) NOTE1))))))
 13:55             --- [RECOGNIZE by RULE386] --->
                   (THEN $CONSIDER-PROP HSM1 CHOICE-SETS
                         (LAMBDA (I1)
                          (IF (= I1 1) (TONES) (USABLE-SUCCESSORS-OF (PROJECT1 (- I1 1))))))
(DEFINE USABLE-SUCCESSORS-OF (NOTE2)
        =
        (SET-OF NOTE1 (TONES) (USABLE-INTERVAL! (INTERVAL NOTE2 NOTE1))))

 13:56             --- [SIMPLIFY by RULE334] --->
                   HSM1
(HSM1 <- CHOICE-SETS :
      (LAMBDA (I1)
       (IF (= I1 1) (TONES) (USABLE-SUCCESSORS-OF (PROJECT1 (- I1 1))))))

> 13:57            --- [by RULE312] --->
     (CONSIDER-PROP
      (LAMBDA (PROJECT1)
       (AND [FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                   (USABLE-INTERVAL! INTERVAL1)]
            [=< (MELODIC-RANGE PROJECT1) (MAJOR 10)]
            [= (#OCCURRENCES (CLIMAX PROJECT1) PROJECT1) 1])))

                                 (= (#OCCURRENCES (CLIMAX PROJECT1) PROJECT1) 1)
| 13:58                          --- [RESTRICT by RULE256] --->
                                 (AND [= (CLIMAX PROJECT1) CLIMAX1]
                                      [= (#OCCURRENCES CLIMAX1 PROJECT1) 1])
(ASSUME (SELECT CLIMAX1 (RANGE (CLIMAX PROJECT1))))

                            (AND [FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                                         (USABLE-INTERVAL! INTERVAL1)]
                                 [=< (MELODIC-RANGE PROJECT1) (MAJOR 10)]
                                 [AND [= (CLIMAX PROJECT1) CLIMAX1]
                                      [= (#OCCURRENCES CLIMAX1 PROJECT1) 1]])
```

```
| 13:59                    --- [SIMPLIFY by RULE177] --->
                           (AND [= (CLIMAX PROJECT1) CLIMAX1]
                                [= (#OCCURRENCES CLIMAX1 PROJECT1) 1]
                                [FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                                        (USABLE-INTERVAL! INTERVAL1)]
                                [=< (MELODIC-RANGE PROJECT1) (MAJOR 10)])

                                   (CLIMAX PROJECT1)
| 13:60                            --- [ELABORATE by RULE124] --->
                                   (HIGHEST PROJECT1)

| 13:61                            --- [ELABORATE by RULE124] --->
                                   (THE C1 PROJECT1 (FORALL X1 PROJECT1 (NOT (HIGHER X1 C1))))

                                (= (THE C1 PROJECT1 (FORALL X1 PROJECT1 (NOT (HIGHER X1 C1))))
                                   CLIMAX1)
| 13:62                         --- [REMOVE-QUANT by RULE131] --->
                                (AND [IN CLIMAX1 PROJECT1]
                                     [FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1))])

                           (AND [AND [IN CLIMAX1 PROJECT1]
                                     [FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1))]]
                                [= (#OCCURRENCES CLIMAX1 PROJECT1) 1]
                                [FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                                        (USABLE-INTERVAL! INTERVAL1)]
                                [=< (MELODIC-RANGE PROJECT1) (MAJOR 10)])
| 13:63                    --- [SIMPLIFY by RULE177] --->
                           (AND [IN CLIMAX1 PROJECT1]
                                [FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1))]
                                [= (#OCCURRENCES CLIMAX1 PROJECT1) 1]
                                [FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                                        (USABLE-INTERVAL! INTERVAL1)]
                                [=< (MELODIC-RANGE PROJECT1) (MAJOR 10)])

    (CONSIDER-PROP
     (LAMBDA (PROJECT1)
      (AND [IN CLIMAX1 PROJECT1]
           [FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1))]
           [= (#OCCURRENCES CLIMAX1 PROJECT1) 1]
           [FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                   (USABLE-INTERVAL! INTERVAL1)]
           [=< (MELODIC-RANGE PROJECT1) (MAJOR 10)])))
< 13:64
    --- [SUCCEED by RULE313] --->
          (ACHIEVE HSM1)
(HSM1 <- SOLUTION-TEST :
      (LAMBDA (PROJECT1)
       (AND [IN CLIMAX1 PROJECT1]
            [FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1))]
            [= (#OCCURRENCES CLIMAX1 PROJECT1) 1]
            [FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                    (USABLE-INTERVAL! INTERVAL1)]
            [=< (MELODIC-RANGE PROJECT1) (MAJOR 10)])))

                   HSM1
> 13:65            --- [by RULE323] --->
    (CONSIDER-PROP
     (AND [AND [LAMBDA (PROJECT1)
                (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                        (USABLE-INTERVAL! INTERVAL1))]
               [LAMBDA (PROJECT1) (=< (MELODIC-RANGE PROJECT1) (MAJOR 10))]]
          [LAMBDA (PROJECT1)
           (=> (=> (FORALL X1 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))
                               (NOT (HIGHER X1 CLIMAX1)))
                       (FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1))))
               (FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1))))]))

                   (AND [AND [LAMBDA (PROJECT1)
                              (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                                      (USABLE-INTERVAL! INTERVAL1))]
                             [LAMBDA (PROJECT1) (=< (MELODIC-RANGE PROJECT1) (MAJOR 10))]]
                        [LAMBDA (PROJECT1)
                         (=> (=> (FORALL X1 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))
                                             (NOT (HIGHER X1 CLIMAX1)))
                                     (FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1))))
                             (FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1))))])
| 13:66            --- [SIMPLIFY by RULE177] --->
                   (AND [LAMBDA (PROJECT1)
                         (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                                 (USABLE-INTERVAL! INTERVAL1))]
                        [LAMBDA (PROJECT1) (=< (MELODIC-RANGE PROJECT1) (MAJOR 10))]
                        [LAMBDA (PROJECT1)
                         (=> (=> (FORALL X1 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))
                                          (NOT (HIGHER X1 CLIMAX1)))
                                 (FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1))))
                             (FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1))))]

                                     (=> (FORALL X1 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))
                                                 (NOT (HIGHER X1 CLIMAX1)))
                                         (FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1))))
|> 13:67                             --- [REDUCE by RULE379] --->
                               (SHOW (SUBSET PROJECT1 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))))

                                     (SUBSET PROJECT1 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))))
|| 13:68                             --- [EVAL by RULE346] --->
                                     T

                               (SHOW T)
|< 13:69                       --- [SUCCEED by RULE53] --->
    (CONSIDER-PROP
     (AND [LAMBDA (PROJECT1)
           (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                   (USABLE-INTERVAL! INTERVAL1))]
          [LAMBDA (PROJECT1) (=< (MELODIC-RANGE PROJECT1) (MAJOR 10))]
          [LAMBDA (PROJECT1)
           (=> T (FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1))))]))

                                 (=> T (FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1))))
| 13:70                          --- [SIMPLIFY by RULE12] --->
                                 (FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1)))

    (CONSIDER-PROP
     (AND [LAMBDA (PROJECT1)
           (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                   (USABLE-INTERVAL! INTERVAL1))]
          [LAMBDA (PROJECT1) (=< (MELODIC-RANGE PROJECT1) (MAJOR 10))]
          [LAMBDA (PROJECT1) (FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1)))]))
< 13:71
      --- [SUCCEED by RULE313] --->
          (ACHIEVE HSM1)
(HSM1 <- PATH-TEST :
      (AND [LAMBDA (PROJECT1)
            (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                     (USABLE-INTERVAL! INTERVAL1))]
          [LAMBDA (PROJECT1) (=< (MELODIC-RANGE PROJECT1) (MAJOR 10))]
           [LAMBDA (PROJECT1) (FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1)))]))

                   HSM1
> 13:72            --- [by RULE327] --->
              (SHOW (= (FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1)))
                       (FORALL I1 (INDICES-OF PROJECT1) Q2)))

                       (FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1)))
| 13:73                --- [by RULE328] --->
                       (FORALL I5 (INDICES-OF PROJECT1)
                               (NOT (HIGHER (PROJECT1 I5) CLIMAX1)))

                   (= (FORALL I5 (INDICES-OF PROJECT1)
                               (NOT (HIGHER (PROJECT1 I5) CLIMAX1)))
                      (FORALL I1 (INDICES-OF PROJECT1) Q2))
| 13:74            --- [REDUCE by RULE192] --->
                   (= (NOT (HIGHER (PROJECT1 I1) CLIMAX1)) Q2)

| 13:75            --- [EVAL by RULE271] --->
                   T
(Q2 <- BINDING : (NOT (HIGHER (PROJECT1 I1) CLIMAX1)))

              (SHOW T)
< 13:76        --- [SUCCEED by RULE34] --->
          (ACHIEVE (THEN $CONSIDER-PROP HSM1 STEP-TEST
                          (AND T [LAMBDA (I1 NOTE3) (THEN $SUBST NOTE3 (PROJECT1 I1) Q2)])))

                          (AND T [LAMBDA (I1 NOTE3) (THEN $SUBST NOTE3 (PROJECT1 I1) Q2)])
 13:77                    --- [SIMPLIFY by RULE341] --->
                          (LAMBDA (I1 NOTE3) (THEN $SUBST NOTE3 (PROJECT1 I1) Q2))

                                        Q2
 13:78                                  --- [EVAL by RULE127] --->
                                        (NOT (HIGHER (PROJECT1 I1) CLIMAX1))

                                  (THEN $SUBST NOTE3 (PROJECT1 I1)
                                        (NOT (HIGHER (PROJECT1 I1) CLIMAX1)))
```

```
13:79                          --- [SIMPLIFY by RULE331] --->
                               (NOT (HIGHER NOTE3 CLIMAX1))

                (THEN $CONSIDER-PROP HSM1 STEP-TEST
                       (LAMBDA (I1 NOTE3) (NOT (HIGHER NOTE3 CLIMAX1))))
13:80           --- [SIMPLIFY by RULE334] --->
                HSM1
(HSM1 <- STEP-TEST : (LAMBDA (I1 NOTE3) (NOT (HIGHER NOTE3 CLIMAX1))))

> 13:81         --- [by RULE312] --->
   (CONSIDER-PROP
    (LAMBDA (PROJECT1)
     (AND [IN CLIMAX1 PROJECT1]
          [FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1))]
          [= (#OCCURRENCES CLIMAX1 PROJECT1) 1]
          [FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                 (USABLE-INTERVAL! INTERVAL1)]
          [=< (MELODIC-RANGE PROJECT1) (MAJOR 10)])))

                               (= (#OCCURRENCES CLIMAX1 PROJECT1) 1)
|> 13:82                       --- [by RULE268] --->
                    (CONSIDER (= (#OCCURRENCES CLIMAX1 PROJECT1) 1))

                               (= (#OCCURRENCES CLIMAX1 PROJECT1) 1)
|| 13:83                       --- [ELABORATE by RULE340] --->
                               (AND [>= (#OCCURRENCES CLIMAX1 PROJECT1) 1]
                                    [>= 1 (#OCCURRENCES CLIMAX1 PROJECT1)])

                                       (#OCCURRENCES CLIMAX1 PROJECT1)
|| 13:84                               --- [ELABORATE by RULE124] --->
                                       (# (SET-OF X2 PROJECT1 (= X2 CLIMAX1)))

                                   (>= (# (SET-OF X2 PROJECT1 (= X2 CLIMAX1))) 1)
|| 13:85                           --- [RECOGNIZE by RULE387] --->
                                   (EXISTS X2 PROJECT1 (= X2 CLIMAX1))

|| 13:86                           --- [RECOGNIZE by RULE292] --->
                                   (IN CLIMAX1 PROJECT1)

                                   (>= 1 (#OCCURRENCES CLIMAX1 PROJECT1))
|| 13:87                           --- [by RULE153] --->
                                   (=< (#OCCURRENCES CLIMAX1 PROJECT1) 1)

                    (CONSIDER (AND [IN CLIMAX1 PROJECT1]
                                   [=< (#OCCURRENCES CLIMAX1 PROJECT1) 1]))
|< 13:88            --- [by RULE269] --->
   (CONSIDER-PROP
    (LAMBDA (PROJECT1)
     (AND [IN CLIMAX1 PROJECT1]
          [FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1))]
          [AND [IN CLIMAX1 PROJECT1]
               [=< (#OCCURRENCES CLIMAX1 PROJECT1) 1]]
          [FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                 (USABLE-INTERVAL! INTERVAL1)]
          [=< (MELODIC-RANGE PROJECT1) (MAJOR 10)])))

                         (AND [IN CLIMAX1 PROJECT1]
                              [FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1))]
                              [AND [IN CLIMAX1 PROJECT1]
                                   [=< (#OCCURRENCES CLIMAX1 PROJECT1) 1]]
                              [FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                                     (USABLE-INTERVAL! INTERVAL1)]
                              [=< (MELODIC-RANGE PROJECT1) (MAJOR 10)])
 | 13:89                 --- [SIMPLIFY by RULE177] --->
                         (AND [IN CLIMAX1 PROJECT1]
                              [=< (#OCCURRENCES CLIMAX1 PROJECT1) 1]
                              [IN CLIMAX1 PROJECT1]
                              [FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1))]
                              [FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                                     (USABLE-INTERVAL! INTERVAL1)]
                              [=< (MELODIC-RANGE PROJECT1) (MAJOR 10)])

 | 13:90                 --- [SIMPLIFY by RULE188] --->
                         (AND [IN CLIMAX1 PROJECT1]
                              [=< (#OCCURRENCES CLIMAX1 PROJECT1) 1]
                              [FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1))]
                              [FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                                     (USABLE-INTERVAL! INTERVAL1)]
                              [=< (MELODIC-RANGE PROJECT1) (MAJOR 10)])

    (CONSIDER-PROP
     (LAMBDA (PROJECT1)
      (AND [IN CLIMAX1 PROJECT1]
           [=< (#OCCURRENCES CLIMAX1 PROJECT1) 1]
           [FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1))]
           [FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                  (USABLE-INTERVAL! INTERVAL1)]
           [=< (MELODIC-RANGE PROJECT1) (MAJOR 10)]))
< 13:91
    --- [SUCCEED by RULE313] --->
          (ACHIEVE HSM1)
(HSM1 <- SOLUTION-TEST :
      (LAMBDA (PROJECT1)
       (AND [IN CLIMAX1 PROJECT1]
            [=< (#OCCURRENCES CLIMAX1 PROJECT1) 1]
            [FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1))]
            [FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                   (USABLE-INTERVAL! INTERVAL1)]
            [=< (MELODIC-RANGE PROJECT1) (MAJOR 10)]))

                   HSM1
> 13:92            --- [by RULE323] --->
    (CONSIDER-PROP
     (AND [AND [LAMBDA (PROJECT1)
                (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                        (USABLE-INTERVAL! INTERVAL1))]
               [LAMBDA (PROJECT1) (=< (MELODIC-RANGE PROJECT1) (MAJOR 10))]
               [LAMBDA (PROJECT1) (FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1)))]]
          [LAMBDA (PROJECT1)
           (=> (=> (=< (#OCCURRENCES CLIMAX1 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))))
                       1)
                   (=< (#OCCURRENCES CLIMAX1 PROJECT1) 1))
               (=< (#OCCURRENCES CLIMAX1 PROJECT1) 1))]]))

                                   (=> (=< (#OCCURRENCES CLIMAX1 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))))
                                           1)
                                       (=< (#OCCURRENCES CLIMAX1 PROJECT1) 1))
|> 13:93                           --- [REDUCE by RULE383] --->
                             (SHOW (=< (#OCCURRENCES CLIMAX1 PROJECT1)
                                       (#OCCURRENCES CLIMAX1 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))))))

                                       (#OCCURRENCES CLIMAX1 PROJECT1)
|| 13:94                               --- [ELABORATE by RULE124] --->
                                       (# (SET-OF X3 PROJECT1 (= X3 CLIMAX1)))

                                       (#OCCURRENCES CLIMAX1 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))))
|| 13:95                               --- [ELABORATE by RULE124] --->
                                       (# (SET-OF X4 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))
                                                  (= X4 CLIMAX1)))

                                   (=< (# (SET-OF X3 PROJECT1 (= X3 CLIMAX1)))
                                       (# (SET-OF X4 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))
                                                  (= X4 CLIMAX1))))
||> 13:96                          --- [REDUCE by RULE30] --->
                             (SHOW (SUBSET (SET-OF X3 PROJECT1 (= X3 CLIMAX1))
                                           (SET-OF X4 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))
                                                   (= X4 CLIMAX1))))

                                   (SUBSET (SET-OF X3 PROJECT1 (= X3 CLIMAX1))
                                           (SET-OF X4 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))
                                                   (= X4 CLIMAX1)))
|||> 13:97                         --- [REDUCE by RULE388] --->
                             (SHOW (SUBSET PROJECT1 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))))

                                   (SUBSET PROJECT1 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))))
|||| 13:98                         --- [EVAL by RULE346] --->
                                   T

                             (SHOW T)
|||< 13:99                   --- [SUCCEED by RULE53] --->
                             (SHOW T)

||< 13:100                   --- [SUCCEED by RULE53] --->
                             (SHOW T)

|< 13:101                    --- [SUCCEED by RULE53] --->
    (CONSIDER-PROP
     (AND [AND [LAMBDA (PROJECT1)
                (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                        (USABLE-INTERVAL! INTERVAL1))]
               [LAMBDA (PROJECT1) (=< (MELODIC-RANGE PROJECT1) (MAJOR 10))]
               [LAMBDA (PROJECT1) (FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1)))]]
```

```
        [LAMBDA (PROJECT1) (=> T (=< (#OCCURRENCES CLIMAX1 PROJECT1) 1))]]))

                             (=> T (=< (#OCCURRENCES CLIMAX1 PROJECT1) 1))
| 13:102                     --- [SIMPLIFY by RULE12] --->
                             (=< (#OCCURRENCES CLIMAX1 PROJECT1) 1)

                    (AND [AND [LAMBDA (PROJECT1)
                                (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                                        (USABLE-INTERVAL! INTERVAL1))]
                              [LAMBDA (PROJECT1) (=< (MELODIC-RANGE PROJECT1) (MAJOR 10))]
                              [LAMBDA (PROJECT1) (FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1)))]]
                         [LAMBDA (PROJECT1) (=< (#OCCURRENCES CLIMAX1 PROJECT1) 1)])
| 13:103            --- [SIMPLIFY by RULE177] --->
                    (AND [LAMBDA (PROJECT1)
                           (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                                   (USABLE-INTERVAL! INTERVAL1))]
                         [LAMBDA (PROJECT1) (=< (MELODIC-RANGE PROJECT1) (MAJOR 10))]
                         [LAMBDA (PROJECT1) (FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1)))]
                         [LAMBDA (PROJECT1) (=< (#OCCURRENCES CLIMAX1 PROJECT1) 1)])

   (CONSIDER-PROP
    (AND [LAMBDA (PROJECT1)
           (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                   (USABLE-INTERVAL! INTERVAL1))]
         [LAMBDA (PROJECT1) (=< (MELODIC-RANGE PROJECT1) (MAJOR 10))]
         [LAMBDA (PROJECT1) (FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1)))]
         [LAMBDA (PROJECT1) (=< (#OCCURRENCES CLIMAX1 PROJECT1) 1)]))
< 13:104
   --- [SUCCEED by RULE313] --->
         (ACHIEVE HSM1)
(HSM1 <- PATH-TEST :
      (AND [LAMBDA (PROJECT1)
             (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                     (USABLE-INTERVAL! INTERVAL1))]
           [LAMBDA (PROJECT1) (=< (MELODIC-RANGE PROJECT1) (MAJOR 10))]
           [LAMBDA (PROJECT1) (FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1)))]
           [LAMBDA (PROJECT1) (=< (#OCCURRENCES CLIMAX1 PROJECT1) 1)]))

                    HSM1
 13:106             --- [REFINE by RULE389] --->
                    (THEN $CONSIDER-PROP HSM1 STEP-TEST
                          (LAMBDA (I1 NOTE3)
                           (AND [NOT (HIGHER NOTE3 CLIMAX1)]
                                [=< (#OCCURRENCES CLIMAX1 (APPEND PROJECT1 (LIST NOTE3))) 1])))

                                       (#OCCURRENCES CLIMAX1 (APPEND PROJECT1 (LIST NOTE3)))
 13:106                                --- [DISTRIBUTE by RULE284] --->
                                       (+ (#OCCURRENCES CLIMAX1 PROJECT1)
                                          (#OCCURRENCES CLIMAX1 (LIST NOTE3)))

                                   (=< (+ (#OCCURRENCES CLIMAX1 PROJECT1)
                                          (#OCCURRENCES CLIMAX1 (LIST NOTE3)))
                                       1)
 13:107                            --- [REDUCE by RULE301] --->
                                   (IF (= CLIMAX1 NOTE3)
                                       (=< (#OCCURRENCES CLIMAX1 PROJECT1) 0)
                                       (=< (#OCCURRENCES CLIMAX1 PROJECT1) 1))

                    (THEN $CONSIDER-PROP HSM1 STEP-TEST
                          (LAMBDA (I1 NOTE3)
                           (AND [NOT (HIGHER NOTE3 CLIMAX1)]
                                [IF (= CLIMAX1 NOTE3)
                                    (=< (#OCCURRENCES CLIMAX1 PROJECT1) 0)
                                    (=< (#OCCURRENCES CLIMAX1 PROJECT1) 1)])))
 13:108             --- [EVAL by RULE390] --->
                    (THEN $CONSIDER-PROP HSM1 STEP-TEST
                          (LAMBDA (I1 NOTE3)
                           (AND [NOT (HIGHER NOTE3 CLIMAX1)]
                                [IF (= CLIMAX1 NOTE3) (=< (#OCCURRENCES CLIMAX1 PROJECT1) 0) T])))

                                   (IF (= CLIMAX1 NOTE3) (=< (#OCCURRENCES CLIMAX1 PROJECT1) 0) T)
 13:109                            --- [RECOGNIZE by RULE391] --->
                                   (OR [NOT (= CLIMAX1 NOTE3)]
                                       [=< (#OCCURRENCES CLIMAX1 PROJECT1) 0])

                                       (=< (#OCCURRENCES CLIMAX1 PROJECT1) 0)
> 13:110                               --- [by RULE288] --->
                             (CONSIDER (=< (#OCCURRENCES CLIMAX1 PROJECT1) 0))

                                       (=< (#OCCURRENCES CLIMAX1 PROJECT1) 0)
| 13:111                               --- [RECOGNIZE by RULE392] --->
```

```
                                         (= (#OCCURRENCES CLIMAX1 PROJECT1) 0)

                                            (#OCCURRENCES CLIMAX1 PROJECT1)
| 13:112                                    --- [ELABORATE by RULE124] --->
                                            (# (SET-OF X5 PROJECT1 (= X5 CLIMAX1)))

                                         (= (# (SET-OF X5 PROJECT1 (= X5 CLIMAX1))) 0)
| 13:113                                 --- [RECOGNIZE by RULE291] --->
                                         (NOT (EXISTS X5 PROJECT1 (= X5 CLIMAX1)))

                                              (EXISTS X5 PROJECT1 (= X5 CLIMAX1))
| 13:114                                      --- [RECOGNIZE by RULE292] --->
                                              (IN CLIMAX1 PROJECT1)

                               (CONSIDER (NOT (IN CLIMAX1 PROJECT1)))
< 13:115                       --- [by RULE269] --->
          (ACHIEVE (THEN $CONSIDER-PROP HSM1 STEP-TEST
                        (LAMBDA (I1 NOTE3)
                         (AND [NOT (HIGHER NOTE3 CLIMAX1)]
                              [OR [NOT (= CLIMAX1 NOTE3)] [NOT (IN CLIMAX1 PROJECT1)]]))))

                  (THEN $CONSIDER-PROP HSM1 STEP-TEST
                        (LAMBDA (I1 NOTE3)
                         (AND [NOT (HIGHER NOTE3 CLIMAX1)]
                              [OR [NOT (= CLIMAX1 NOTE3)] [NOT (IN CLIMAX1 PROJECT1)]])))
 13:118           --- [SIMPLIFY by RULE334] --->
                  HSM1
(HSM1 <- STEP-TEST :
      (LAMBDA (I1 NOTE3)
       (AND [NOT (HIGHER NOTE3 CLIMAX1)]
            [OR [NOT (= CLIMAX1 NOTE3)] [NOT (IN CLIMAX1 PROJECT1)]])))

          [Final expression:]
          (ACHIEVE HSM1)

(ASSUMING (SELECT CLIMAX1 (RANGE (CLIMAX PROJECT1))))

(ASSUMING (SUBSET PROJECT1 (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))))
NIL
<7>p hsm1

(HSM1 WITH (PROBLEM : (LEGAL-CANTUS! (CANTUS)))
      (OBJECT : (CANTUS))
      (CHOICE-SEQ :
                  (EACH I1 (LB:UB 1 (CANTUS-LENGTH))
                        (CHOOSE (NOTE I1) (TONES) (NOTE I1))))
      (SEQUENCE : NOTE)
      (CHOICES : (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH))))
      (CHOICE-SETS :
                   (LAMBDA (I1)
                    (IF (= I1 1)
                        (TONES)
                        (USABLE-SUCCESSORS-OF (PROJECT1 (- I1 1))))))
      (INDICES : (LB:UB 1 (CANTUS-LENGTH)))
      (INDEX : I1)
      (VARIABLES : NIL)
      (BINDINGS : NIL)
      (INITIAL-PATH : NIL)
      (COMPLETION-TEST :
       (LAMBDA (PROJECT1) (IN (# PROJECT1) (LB:UB 8 16))))
      (REFORMULATED-PROBLEM :
       ((LAMBDA (PROJECT1)
         (AND [IN (# PROJECT1) (LB:UB 8 16)]
              [FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                      (USABLE-INTERVAL! INTERVAL1)]
              [=< (MELODIC-RANGE PROJECT1) (MAJOR 10)]
              [= (#OCCURRENCES (CLIMAX PROJECT1) PROJECT1) 1]))
        (PROJECT NOTE (LB:UB 1 (CANTUS-LENGTH)))))
      (PATH : PROJECT1)
      (STEP-ORDER : NIL)
      (STEP-TEST :
                 (LAMBDA (I1 NOTE3)
                  (AND [NOT (HIGHER NOTE3 CLIMAX1)]
                       [OR [NOT (= CLIMAX1 NOTE3)]
                           [NOT (IN CLIMAX1 PROJECT1)]])))
      (PATH-ORDER : NIL)
      (PATH-TEST :
                 (AND [LAMBDA (PROJECT1)
                       (FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                               (USABLE-INTERVAL! INTERVAL1))]
                      [LAMBDA (PROJECT1)
```

```
                       (=< (MELODIC-RANGE PROJECT1) (MAJOR 10))]
                      [LAMBDA (PROJECT1)
                       (FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1)))]
                      [LAMBDA (PROJECT1)
                       (=< (#OCCURRENCES CLIMAX1 PROJECT1) 1)]))
       (SOLUTION-TEST :
        (LAMBDA (PROJECT1)
         (AND [IN CLIMAX1 PROJECT1]
              [=< (#OCCURRENCES CLIMAX1 PROJECT1) 1]
              [FORALL X1 PROJECT1 (NOT (HIGHER X1 CLIMAX1))]
              [FORALL INTERVAL1 (INTERVALS-OF PROJECT1)
                      (USABLE-INTERVAL! INTERVAL1)]
              [=< (MELODIC-RANGE PROJECT1) (MAJOR 10)]))))
HSM1
<8>pp usable-successors-of

(DE USABLE-SUCCESSORS-OF (NOTE2)
    (SET-OF NOTE1 (TONE.) (USABLE-INTERVAL! (INTERVAL NOTE2 NOTE1))))

NIL
<9>recordfile

RECORD FILE DSK: (D13007 . PZG) CLOSED 07-DEC-80 19:42:51
```

D.14. DERIV14: Achieve Unique Climax

```
RECORD FILE DSK: (D14007 . PZG) OPENED 07-DEC-80 19:46:59
NIL
<23>p-links nil

(DERIV14 STARTED "01-JUL-80 03:06:01" #LINKS NIL --DOMAIN: MUSIC
         --PROBLEM: (ACHIEVE (= (#OCCURRENCES (CLIMAX CANTUS) CANTUS) 1)))


         [Initial expression:]
         (ACHIEVE (= (#OCCURRENCES (CLIMAX CANTUS) CANTUS) 1))

 14:1    --- [by RULE266] --->
         (FORALL T1 (LB:UB 0 (- (# CANTUS) 1))
                 ((ACHIEVE (= (#OCCURRENCES (CLIMAX (LAMBDA (J1) (PREFIX CANTUS J1)))
                                             (LAMBDA (J1) (PREFIX CANTUS J1)))
                              1))
                  T1))

                 (ACHIEVE (= (#OCCURRENCES (CLIMAX (LAMBDA (J1) (PREFIX CANTUS J1)))
                                            (LAMBDA (J1) (PREFIX CANTUS J1)))
                             1))
> 14:2           --- [by RULE268] --->
        (CONSIDER (ACHIEVE (= (#OCCURRENCES (CLIMAX (LAMBDA (J1) (PREFIX CANTUS J1)))
                                             (LAMBDA (J1) (PREFIX CANTUS J1)))
                              1)))

                                                  (LAMBDA (J1) (PREFIX CANTUS J1))
| 14:3                                            --- [RECOGNIZE by RULE267] --->
                                                  CANTUS-1

                                        (LAMBDA (J1) (PREFIX CANTUS J1))
| 14:4                                  --- [RECOGNIZE by RULE267] --->
                                        CANTUS-1

                 (ACHIEVE (= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1))
| 14:5           --- [by RULE282] --->
                 (NEXT (= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1))

| 14:6           --- [REDUCE by RULE283] --->
                 (= (NEXT (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1)) 1)

                    (NEXT (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1))
| 14:7              --- [REDUCE by RULE283] --->
                    (#OCCURRENCES (NEXT (CLIMAX CANTUS-1)) (NEXT CANTUS-1))

                 (= (#OCCURRENCES (NEXT (CLIMAX CANTUS-1)) (NEXT CANTUS-1)) 1)
| 14:8           --- [DISTRIBUTE by RULE276] --->
                 (OR [AND [NOT (CHANGE (CLIMAX CANTUS-1))]
                           [= (#OCCURRENCES (CLIMAX CANTUS-1) (NEXT CANTUS-1)) 1]]
                       [AND [CHANGE (CLIMAX CANTUS-1)]
                           [= (#OCCURRENCES (NEXT (CLIMAX CANTUS-1)) (NEXT CANTUS-1)) 1]])

                           (= (#OCCURRENCES (CLIMAX CANTUS-1) (NEXT CANTUS-1)) 1)
|> 14:9                    --- [by RULE268] --->
                  (CONSIDER (= (#OCCURRENCES (CLIMAX CANTUS-1) (NEXT CANTUS-1)) 1))

                                            (NEXT CANTUS-1)
|| 14:10                                    --- [ELABORATE by RULE124] --->
                                            (LAMBDA (J) (CANTUS-1 (+ J 1)))

                                                    (CANTUS-1 (+ J 1))
|| 14:11                                            --- [ELABORATE by RULE124] --->
                                                    (PREFIX CANTUS (+ J 1))

|| 14:12                                            --- [ELABORATE by RULE274] --->
                                                    (APPEND (PREFIX CANTUS J) (LIST (NTH CANTUS (+ J 1))))

                                            (LAMBDA (J) (APPEND (PREFIX CANTUS J) (LIST (NTH CANTUS (+ J 1)))))
|| 14:13                                    --- [by RULE393] --->
                                            (APPEND (LAMBDA (J) (PREFIX CANTUS J))
                                                    (LAMBDA (J) (LIST (NTH CANTUS (+ J 1)))))

                                                    (LAMBDA (J) (PREFIX CANTUS J))
|| 14:14                                            --- [RECOGNIZE by RULE267] --->
                                                    CANTUS-1

                                                                  (NTH CANTUS (+ J 1))
|| 14:15                                                          --- [RECOGNIZE by RULE123] --->
                                                                  (NOTE (+ J 1))

                                                    (LAMBDA (J) (LIST (NOTE (+ J 1))))
|| 14:16                                            --- [by RULE393] --->
                                                    (LIST (LAMBDA (J) (NOTE (+ J 1))))

                                                          (LAMBDA (J) (NOTE (+ J 1)))
|| 14:17                                                  --- [RECOGNIZE by RULE123] --->
                                                          (NEXT NOTE)

                              (#OCCURRENCES (CLIMAX CANTUS-1)
                                            (APPEND CANTUS-1 (LIST (NEXT NOTE))))
|| 14:18                      --- [DISTRIBUTE by RULE284] --->
                              (+ (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1)
                                 (#OCCURRENCES (CLIMAX CANTUS-1) (LIST (NEXT NOTE))))

                                 (#OCCURRENCES (CLIMAX CANTUS-1) (LIST (NEXT NOTE)))
|| 14:19                         --- [ELABORATE by RULE124] --->
                                 (# (SET-OF X1 (LIST (NEXT NOTE)) (= X1 (CLIMAX CANTUS-1))))

                                    (SET-OF X1 (LIST (NEXT NOTE)) (= X1 (CLIMAX CANTUS-1)))
|| 14:20                            --- [DISTRIBUTE by RULE121] --->
                                    (UNION (IF (= (NEXT NOTE) (CLIMAX CANTUS-1)) (SET (NEXT NOTE)) NIL))

|| 14:21                            --- [SIMPLIFY by RULE178] --->
                                    (IF (= (NEXT NOTE) (CLIMAX CANTUS-1)) (SET (NEXT NOTE)) NIL)

                                 (# (IF (= (NEXT NOTE) (CLIMAX CANTUS-1)) (SET (NEXT NOTE)) NIL))
|| 14:22                         --- [DISTRIBUTE by RULE287] --->
                                 (IF (= (NEXT NOTE) (CLIMAX CANTUS-1))
                                     (# (SET (NEXT NOTE)))
                                     (# NIL))

                                     (# NIL)
|| 14:23                             --- [SIMPLIFY by RULE289] --->
                                     0

                                     (# (SET (NEXT NOTE)))
|| 14:24                             --- [COMPUTE by RULE288] --->
                                     1

                           (= (+ (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1)
                                 (IF (= (NEXT NOTE) (CLIMAX CANTUS-1)) 1 0))
                              1)
|| 14:25                   --- [DISTRIBUTE by RULE287] --->
                           (IF (= (NEXT NOTE) (CLIMAX CANTUS-1))
                               (= (+ (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1) 1)
                               (= (+ (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 0) 1))

                                  (+ (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 0)
|| 14:26                          --- [SIMPLIFY by RULE343] --->
```

```
                                   (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1)

                                (= (+ (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1) 1)
|| 14:27                        --- [REDUCE by RULE290] --->
                                (= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 0)

||> 14:28                       --- [by RULE268] --->
                      (CONSIDER (= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 0))

                                   (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1)
||| 14:29                          --- [ELABORATE by RULE124] --->
                                   (# (SET-OF X2 CANTUS-1 (= X2 (CLIMAX CANTUS-1))))

                                (= (# (SET-OF X2 CANTUS-1 (= X2 (CLIMAX CANTUS-1)))) 0)
||| 14:30                       --- [RECOGNIZE by RULE291] --->
                                (NOT (EXISTS X2 CANTUS-1 (= X2 (CLIMAX CANTUS-1))))

                                     (EXISTS X2 CANTUS-1 (= X2 (CLIMAX CANTUS-1)))
||| 14:31                            --- [RECOGNIZE by RULE292] --->
                                     (IN (CLIMAX CANTUS-1) CANTUS-1)

                                         (CLIMAX CANTUS-1)
||| 14:32                                --- [ELABORATE by RULE124] --->
                                         (HIGHEST CANTUS-1)

                                     (IN (HIGHEST CANTUS-1) CANTUS-1)
||| 14:33                            --- [SIMPLIFY by RULE293] --->
                                     T

                                (NOT T)
||| 14:34                       --- [COMPUTE by RULE236] --->
                                NIL

                      (CONSIDER NIL)
||< 14:35             --- [by RULE269] --->
                  (CONSIDER (IF (= (NEXT NOTE) (CLIMAX CANTUS-1))
                                NIL (= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1)))

                            (IF (= (NEXT NOTE) (CLIMAX CANTUS-1))
                                NIL (= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1))
|| 14:36                    --- [RECOGNIZE by RULE294] --->
                            (AND [= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1]
                                 [NOT (= (NEXT NOTE) (CLIMAX CANTUS-1))])

                  (CONSIDER (AND [= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1]
                                 [NOT (= (NEXT NOTE) (CLIMAX CANTUS-1))]))
|< 14:37          --- [by RULE269] --->
         (CONSIDER (OR [AND [NOT (CHANGE (CLIMAX CANTUS-1))]
                            [AND [= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1]
                                 [NOT (= (NEXT NOTE) (CLIMAX CANTUS-1))]]]
                       [AND [CHANGE (CLIMAX CANTUS-1)]
                            [= (#OCCURRENCES (NEXT (CLIMAX CANTUS-1)) (NEXT CANTUS-1)) 1]]))

                       (AND [NOT (CHANGE (CLIMAX CANTUS-1))]
                            [AND [= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1]
                                 [NOT (= (NEXT NOTE) (CLIMAX CANTUS-1))]]])
| 14:38                --- [SIMPLIFY by RULE177] --->
                       (AND [= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1]
                            [NOT (= (NEXT NOTE) (CLIMAX CANTUS-1))]
                            [NOT (CHANGE (CLIMAX CANTUS-1))])

|> 14:39               --- [by RULE268] --->
              (CONSIDER (AND [= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1]
                             [NOT (= (NEXT NOTE) (CLIMAX CANTUS-1))]
                             [NOT (CHANGE (CLIMAX CANTUS-1))]))

                                 (CHANGE (CLIMAX CANTUS-1))
||> 14:40                        --- [by RULE268] --->
                       (CONSIDER (CHANGE (CLIMAX CANTUS-1)))

                                         (CLIMAX CANTUS-1)
||| 14:41                                 -- [ELABORATE by RULE124] --->
                                         (HIGHEST CANTUS-1)

                                 (CHANGE (HIGHEST CANTUS-1))
||| 14:42                        --- [REDUCE by RULE277] --->
                                 (HIGHER (HIGHEST (CHANGE CANTUS-1)) (HIGHEST CANTUS-1))

                                         (HIGHEST CANTUS-1)
||| 14:43                                --- [RECOGNIZE by RULE123] --->
                                         (CLIMAX CANTUS-1)
```

```
                                                   (CHANGE CANTUS-1)
||| 14:44                                          --- [REDUCE by RULE279] --->
                                                   (LIST (NEXT NOTE))

                                          (HIGHEST (LIST (NEXT NOTE)))
||| 14:46                                 --- [SIMPLIFY by RULE281] --->
                                          (NEXT NOTE)

                        (CONSIDER (HIGHER (NEXT NOTE) (CLIMAX CANTUS-1)))
||< 14:46               --- [by RULE269] --->
             (CONSIDER (AND [= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1]
                             [NOT (= (NEXT NOTE) (CLIMAX CANTUS-1))]
                             [NOT (HIGHER (NEXT NOTE) (CLIMAX CANTUS-1))]))

                        (AND [= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1]
                             [NOT (= (NEXT NOTE) (CLIMAX CANTUS-1))]
                             [NOT (HIGHER (NEXT NOTE) (CLIMAX CANTUS-1))])
|| 14:47                --- [COLLECT by RULE296] --->
                        (AND [= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1]
                             [NOT (OR [HIGHER (NEXT NOTE) (CLIMAX CANTUS-1)]
                                      [= (NEXT NOTE) (CLIMAX CANTUS-1)]])])

                                  (OR [HIGHER (NEXT NOTE) (CLIMAX CANTUS-1)]
                                      [= (NEXT NOTE) (CLIMAX CANTUS-1)])
|| 14:48                          --- [COLLECT by RULE297] --->
                                  ((OR HIGHER =) (NEXT NOTE) (CLIMAX CANTUS-1))

                             (NOT ((OR HIGHER =) (NEXT NOTE) (CLIMAX CANTUS-1)))
|| 14:49                     --- [by RULE298] --->
                             ((NOT (OR HIGHER =)) (NEXT NOTE) (CLIMAX CANTUS-1))

                              (NOT (OR HIGHER =))
|| 14:50                      --- [RECOGNIZE by RULE299] --->
                                    LOWER

             (CONSIDER (AND [= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1]
                             [LOWER (NEXT NOTE) (CLIMAX CANTUS-1)]))
|< 14:51     --- [by RULE269] --->
         (CONSIDER (OR [AND [= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1]
                             [LOWER (NEXT NOTE) (CLIMAX CANTUS-1)]]
                        [AND [CHANGE (CLIMAX CANTUS-1)]
                             [= (#OCCURRENCES (NEXT (CLIMAX CANTUS-1)) (NEXT CANTUS-1)) 1]]))

                        (AND [CHANGE (CLIMAX CANTUS-1)]
                             [= (#OCCURRENCES (NEXT (CLIMAX CANTUS-1)) (NEXT CANTUS-1)) 1])
|> 14:52                --- [by RULE268] --->
             (CONSIDER (AND [CHANGE (CLIMAX CANTUS-1)]
                             [= (#OCCURRENCES (NEXT (CLIMAX CANTUS-1)) (NEXT CANTUS-1)) 1]))

                                                    (CLIMAX CANTUS-1)
|| 14:53                                            --- [ELABORATE by RULE124] --->
                                                    (HIGHEST CANTUS-1)

                             (CHANGE (CLIMAX CANTUS-1))
|| 14:54                     --- [REDUCE by RULE301] --->
                             (HIGHER (HIGHEST (CHANGE CANTUS-1)) (HIGHEST CANTUS-1))

                                              (NEXT (HIGHEST CANTUS-1))
||> 14:55                                     --- [REDUCE by RULE300] --->
                                        (SHOW (HIGHER (HIGHEST (CHANGE CANTUS-1)) (HIGHEST CANTUS-1)))

                                              (HIGHER (HIGHEST (CHANGE CANTUS-1)) (HIGHEST CANTUS-1))
||| 14:56                                     --- [EVAL by RULE302] --->
                                              T

                                        (SHOW T)
||< 14:57                               --- [SUCCEED by RULE34] --->
             (CONSIDER (AND [HIGHER (HIGHEST (CHANGE CANTUS-1)) (HIGHEST CANTUS-1)]
                             [= (#OCCURRENCES (HIGHEST (CHANGE CANTUS-1)) (NEXT CANTUS-1)) 1]))

                                              (CHANGE CANTUS-1)
|| 14:58                                      --- [REDUCE by RULE279] --->
                                              (LIST (NEXT NOTE))

                                     (HIGHEST (LIST (NEXT NOTE)))
|| 14:59                             --- [SIMPLIFY by RULE281] --->
                                     (NEXT NOTE)

                                              (HIGHEST (CHANGE CANTUS-1))
|| 14:60                                      --- [REDUCE by RULE301] --->
```

```
                                    (NEXT NOTE)

                                            (NEXT CANTUS-1)
|| 14:61                                    --- [REDUCE by RULE301] --->
                                            (APPEND CANTUS-1 (LIST (NEXT NOTE)))

                               (#OCCURRENCES (NEXT NOTE) (APPEND CANTUS-1 (LIST (NEXT NOTE))))
|| 14:62                       --- [DISTRIBUTE by RULE284] --->
                               (+ (#OCCURRENCES (NEXT NOTE) CANTUS-1)
                                  (#OCCURRENCES (NEXT NOTE) (LIST (NEXT NOTE))))

                                  (#OCCURRENCES (NEXT NOTE) (LIST (NEXT NOTE)))
|| 14:63                          --- [ELABORATE by RULE124] --->
                                  (# (SET-OF X4 (LIST (NEXT NOTE)) (= X4 (NEXT NOTE))))

                                     (SET-OF X4 (LIST (NEXT NOTE)) (= X4 (NEXT NOTE)))
|| 14:64                             --- [DISTRIBUTE by RULE121] --->
                                     (UNION (IF (= (NEXT NOTE) (NEXT NOTE)) (SET (NEXT NOTE)) NIL))

|| 14:65                             --- [SIMPLIFY by RULE178] --->
                                     (IF (= (NEXT NOTE) (NEXT NOTE)) (SET (NEXT NOTE)) NIL)

                                         (= (NEXT NOTE) (NEXT NOTE))
|| 14:66                                 --- [EVAL by RULE179] --->
                                         T

                                     (IF T (SET (NEXT NOTE)) NIL)
|| 14:67                             --- [SIMPLIFY by RULE185] --->
                                     (SET (NEXT NOTE))

                                  (# (SET (NEXT NOTE)))
|| 14:68                          --- [COMPUTE by RULE288] --->
                                  1

                            (= (+ (#OCCURRENCES (NEXT NOTE) CANTUS-1) 1) 1)
|| 14:69                    --- [REDUCE by RULE290] --->
                            (= (#OCCURRENCES (NEXT NOTE) CANTUS-1) 0)

                               (#OCCURRENCES (NEXT NOTE) CANTUS-1)
|| 14:70                       --- [ELABORATE by RULE124] --->
                               (# (SET-OF X5 CANTUS-1 (= X5 (NEXT NOTE))))

                            (= (# (SET-OF X5 CANTUS-1 (= X5 (NEXT NOTE)))) 0)
|| 14:71                    --- [RECOGNIZE by RULE291] --->
                            (NOT (EXISTS X5 CANTUS-1 (= X5 (NEXT NOTE))))

                                 (EXISTS X5 CANTUS-1 (= X5 (NEXT NOTE)))
|| 14:72                         --- [RECOGNIZE by RULE292] --->
                                 (IN (NEXT NOTE) CANTUS-1)

||> 14:73                        --- [REDUCE by RULE304] --->
                           (SHOW (NOT (HIGHER (NEXT NOTE) (HIGHEST CANTUS-1))))

                                      (HIGHER (NEXT NOTE) (HIGHEST CANTUS-1))
||| 14:74                             --- [EVAL by RULE302] --->
                                      T

                                 (NOT T)
||| 14:75                        --- [COMPUTE by RULE236] --->
                                 NIL

                           (SHOW NIL)
||< 14:76                  --- [SUCCEED by RULE52] --->
             (CONSIDER (AND [HIGHER (NEXT NOTE) (HIGHEST CANTUS-1)] [NOT NIL]))

                            (NOT NIL)
|| 14:77                    --- [COMPUTE by RULE236] --->
                            T

                       (AND [HIGHER (NEXT NOTE) (HIGHEST CANTUS-1)] T)
|| 14:78               --- [SIMPLIFY by RULE343] --->
                       (HIGHER (NEXT NOTE) (HIGHEST CANTUS-1))

                               (HIGHEST CANTUS-1)
|| 14:79                       --- [RECOGNIZE by RULE123] --->
                               (CLIMAX CANTUS-1)

             (CONSIDER (HIGHER (NEXT NOTE) (CLIMAX CANTUS-1)))
|< 14:80     --- [by RULE269] --->
         (CONSIDER (OR [AND [= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1]
                            [LOWER (NEXT NOTE) (CLIMAX CANTUS-1)]]
```

```
                    [HIGHER (NEXT NOTE) (CLIMAX CANTUS-1)]))

          (CONSIDER (OR [AND [= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1]
                              [LOWER (NEXT NOTE) (CLIMAX CANTUS-1)]]
                         [HIGHER (NEXT NOTE) (CLIMAX CANTUS-1)]))
< 14:81   --- [by RULE269] --->
           (FORALL T1 (LB:UB 0 (- (# CANTUS) 1))
                  ((OR [AND [= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1]
                              [LOWER (NEXT NOTE) (CLIMAX CANTUS-1)]]
                         [HIGHER (NEXT NOTE) (CLIMAX CANTUS-1)])
                   T1))

                  ((OR [AND [= (#OCCURRENCES (CLIMAX CANTUS-1) CANTUS-1) 1]
                              [LOWER (NEXT NOTE) (CLIMAX CANTUS-1)]]
                         [HIGHER (NEXT NOTE) (CLIMAX CANTUS-1)])
                   T1)
 14:82             --- [REDUCE by RULE273] --->
                  (OR [AND [= (#OCCURRENCES (CLIMAX (CANTUS-1 T1)) (CANTUS-1 T1)) 1]
                             [LOWER (NEXT (NOTE T1)) (CLIMAX (CANTUS-1 T1))]]
                        [HIGHER (NEXT (NOTE T1)) (CLIMAX (CANTUS-1 T1))])

                                 (NEXT (NOTE T1))
> 14:83                          --- [by RULE268] --->
                        (CONSIDER (NEXT (NOTE T1)))

                                 (NEXT (NOTE T1))
| 14:84                          --- [COLLECT by RULE305] --->
                                 ((NEXT NOTE) T1)

                                  (NEXT NOTE)
| 14:85                           --- [ELABORATE by RULE124] --->
                                  (LAMBDA (J) (NOTE (+ J 1)))

                                 ((LAMBDA (J) (NOTE (+ J 1))) T1)
| 14:86                          --- [SIMPLIFY by RULE213] --->
                                 (NOTE (+ T1 1))

                        (CONSIDER (NOTE (+ T1 1)))
< 14:87                 --- [by RULE269] --->
           (FORALL T1 (LB:UB 0 (- (# CANTUS) 1))
                  (OR [AND [= (#OCCURRENCES (CLIMAX (CANTUS-1 T1)) (CANTUS-1 T1)) 1]
                             [LOWER (NOTE (+ T1 1)) (CLIMAX (CANTUS-1 T1))]]
                        [HIGHER (NOTE (+ T1 1)) (CLIMAX (CANTUS-1 T1))]))

           [Final expression:]
           (FORALL T1 (LB:UB 0 (- (# CANTUS) 1))
                  (OR [AND [= (#OCCURRENCES (CLIMAX (CANTUS-1 T1)) (CANTUS-1 T1)) 1]
                             [LOWER (NOTE (+ T1 1)) (CLIMAX (CANTUS-1 T1))]]
                        [HIGHER (NOTE (+ T1 1)) (CLIMAX (CANTUS-1 T1))]))
NIL recordfile

RECORD FILE DSK: (D14007 . PZG) CLOSED 07-DEC-80 19:51:18
```

Appendix E Operationality

This appendix shows the result of applying a simple recursive procedure for analyzing the operationality of an expression. For each solution derived using FOO, the procedure identifies a set of conditions under which it is operational.

```
RECORD FILE DSK: (OPE319 . QZG) OPENED 20-MAR-81 18:45:10
NIL
<92>for-each d derivations (p-o d)

DERIV2
(ACHIEVE (NOT HSM1))

is operational if
        NOT is ACHIEVABLE
                specifically (NOT HSM1) is DOABLE
        HSM1 is KNOWN


is operational if
        POSSIBLE is COMPUTABLE
                specifically (POSSIBLE (LEGAL P1 C2)) is EVALUABLE
        LEGAL is COMPUTABLE
                specifically (LEGAL P1 C2) is EVALUABLE
        CARDS-PLAYED1 is KNOWN
                specifically (CARDS-PLAYED1 ME) is EVALUABLE
        MY-CARD4 is KNOWN
                specifically (MY-CARD4 : (CARD-OF ME)) is EVALUABLE
        SUIT-LED2 is KNOWN
                specifically (SUIT-LED2 : (SUIT-LED)) is EVALUABLE
        CARDS-PLAYED1 is COMPUTABLE
                specifically (CARDS-PLAYED1 ME) is EVALUABLE
        (CARD-OF (LEADER)) is EVALUABLE
        (PROJECT CARD-OF (PREFIX (PLAYERS) ME)) is EVALUABLE


DERIV3
(UNTIL (PLAYED! QS) (ACHIEVE (LEAD-SPADE ME)))

is operational if
        LEAD-SPADE is ACHIEVABLE
                specifically (LEAD-SPADE ME) is DOABLE


DERIV4
(EVAL (NOT (OR [IN-POT QS] [AT QS HOLE] [HAS-ME QS] [TAKEN QS])))

is operational if
        (AT QS HOLE) is EVALUABLE
        TAKE is OBSERVABLE


DERIV6
```

```
(UNTIL (PLAYED! QS) (ACHIEVE (LEAD-SAFE-SPADE ME)))

is operational if
        LEAD-SAFE-SPADE is ACHIEVABLE
                specifically (LEAD-SAFE-SPADE ME) is DOABLE


DERIV6
(ACHIEVE (=> (AND [IN-SUIT-LED (CARD-OF ME)] [TRICK-HAS-POINTS])
             (LOW (CARD-OF ME))))

is operational if
        (CARD-OF (LEADER)) is EVALUABLE
        (HAVE-POINTS (CARDS-PLAYED)) is EVALUABLE

(ACHIEVE (=> (AND [IN-SUIT-LED (CARD-OF ME)] [TRICK-HAS-POINTS])
             (LOWER (CARD-OF ME) DK)))

is operational if
        (CARD-OF (LEADER)) is EVALUABLE
        (HAVE-POINTS (CARDS-PLAYED)) is EVALUABLE


DERIV7
(ACHIEVE (AND [IN-SUIT-LED (CARD-OF ME)] [HIGH (CARD-OF ME)]))

is operational if
        (CARD-OF (LEADER)) is EVALUABLE


DERIV8
(UNTIL (VOID ME SO) (ACHIEVE (PLAY-SUIT ME SO)))

is operational if
        (VOID ME SO) is EVALUABLE
        SO is KNOWN
        PLAY-SUIT is ACHIEVABLE
                specifically (PLAY-SUIT ME SO) is DOABLE


DERIV9
(FUNCTION-OF (#CARDS-OUT-IN-SUIT SO) DECREASING)

is operational if
        SO is KNOWN
        OUT is COMPUTABLE


DERIV10
(EVAL (- (- 13
            (# (SET-OF X1 (CARDS-IN-SUIT SO)
                          (BEFORE (CURRENT ROUND-IN-PROGRESS) (HAS ME X1)))))
         (# (SET-OF X1 (CARDS-IN-SUIT SO)
                    (WAS-DURING (CURRENT ROUND-IN-PROGRESS)
                                (EXISTS P3 (OPPONENTS ME) (PLAY P3 X1)))))))

is operational if
        (SET-OF X1 (CARDS-IN-SUIT SO) (BEFORE (CURRENT ROUND-IN-PROGRESS) (HAS ME X1))) is EVALUABLE
        SO is KNOWN
        PLAY is OBSERVABLE
                specifically (PLAY P3 X1) is DETECTABLE


DERIV11
(EVAL (AND [BREAKING-SUIT PO] [= (SUIT-LED) SO] [FOLLOWING PO]))

is operational if
        (CARD-OF PO) is EVALUABLE
        PO is KNOWN
        (CARD-OF (LEADER)) is EVALUABLE
        SO is KNOWN
```

```
DERIV12
(EVAL (WAS-DURING (CURRENT ROUND-IN-PROGRESS) (VOID P0 S0)))

is operational if
        VOID is OBSERVABLE
                specifically (VOID P0 S0) is DETECTABLE
        P0 is KNOWN
        S0 is KNOWN


DERIV13
(ACHIEVE HSM1)

is operational if
        HSM1 is EVALUABLE



is operational if
        PROJECT1 is COMPUTABLE
                specifically (PROJECT1 (- I1 1)) is EVALUABLE
        CANTUS-LENGTH is COMPUTABLE
                specifically (CANTUS-LENGTH) is EVALUABLE
        CLIMAX1 is KNOWN
        PROJECT1 is KNOWN
                specifically (PROJECT1 (- I1 1)) is EVALUABLE


DERIV14
(FORALL T1 (LB:UB 0 (- (# CANTUS) 1))
        (OR [AND [= (#OCCURRENCES (CLIMAX (CANTUS-1 T1))
                                   (CANTUS-1 T1)) 1]
                 [LOWER (NOTE (+ T1 1)) (CLIMAX (CANTUS-1 T1))]]
             [HIGHER (NOTE (+ T1 1)) (CLIMAX (CANTUS-1 T1))]))

is operational if
        (# CANTUS) is EVALUABLE
        CLIMAX is COMPUTABLE
                specifical'y (CLIMAX (CANTUS-1 T1)) is EVALUABLE
        CANTUS-1 is COMPUTABLE
                specifically (CANTUS-1 T1) is EVALUABLE
        NOTE is COMPUTABLE
                specifically (NOTE (+ T1 1)) is EVALUABLE


NIL
<93>recordfile

RECORD FILE DSK: (OPE319 . QZG) CLOSED 20-MAR-81 18:47:10
```

For real die-hards, here is the procedure:

```
RECORD FILE DSK: (OPE322 . QZG) OPENED 22-MAR-81 23:07:22
NIL
<33>pp p-o

(DE P-O (N)
    (*** ANALYZE OPERATIONALITY OF N)
    (COND [(N! N) (P-O (E<N N))]
          [(DERIV1 N)
           (MSG T N)
           (MAPCAR (FUNCTION P-O) (GET N 'FINAL-EXPRESSIONS))]
          [(PROG (PL CRITERION VARIABLES FNS-STACK)
                 (SETQ PL (E>O N))
                 (*** (SPRINT PL 1))
                 (IF (NEQ (CADR N) 'WITH) (SPRINT N 1))
                 (MSG T T "is operational")
                 (IF PL (MSG." if"))
                 (FOR-EACH PAIR (SET<LIST PL)
                           (MSG T 8 (CADR PAIR) " is " (CAR PAIR))
                           (FOR-EACH E (EL<FN N (CADR PAIR))
                                     (MSG T 16 "specifically " E " is "
                                          (SELECTQ [CAR PAIR]
                                                   [COMPUTABLE 'EVALUABLE]
                                                   [VISIBLE 'OBSERVABLE]
                                                   [OBSERVABLE 'DETECTABLE]
                                                   [ACHIEVABLE 'DOABLE]
                                                   'EVALUABLE))))
                 (MSG T T))])
    N)

NIL
<34>recordfile

RECORD FILE DSK: (OPE322 . QZG) CLOSED 22-MAR-81 23:07:46
```

References

[Anderson 75]	A. R. Anderson and N. Belnap. *Entailment.* Princeton University Press, Princeton, NJ, 1975.
[Balzer 66]	R. M. Balzer. A mathematical model for performing a complex task in a card game. *Behavioral Science* 2(3):219-236, May, 1966.
[Balzer 77]	R. Balzer, N. Goldman, and D. Wile. Informality in program specifications. In *IJCAI-5*, pages 389-397. Cambridge, MA, 1977.
[Balzer 79]	R. Balzer and N. Goldman. Principles of good software specification and their implications for specification languages. In *Proc. Conf. Specifications Reliable Software*, pages 58ff. Boston, MA, 1979.
[Barstow 77]	D. Barstow. A knowledge-based system for automatic program construction. In *IJCAI-5*, pages 382-388. Cambridge, MA, 1977.
[Berliner 79]	H. Berliner. On the construction of evaluation functions for large domains. In *IJCAI-6*, pages 53-55. 1979.
[Bobrow 77]	D. G. Bobrow and T. Winograd. An overview of KRL, a knowledge representation language. *Cognitive Science* 1(1):3-46, 1977.
[Brachman 79]	R. J. Brachman et al. *Research in Natural Language Understanding.* Technical Report 4274, Bolt Beranek and Newman, 1979.
[Buchanan 69]	B. G. Buchanan, G. Sutherland, and E. A. Feigenbaum. Heuristic Dendral: A program for generating explanatory hypotheses in organic chemistry. In B. Meltzer and D. Michie (editors), *Machine Intelligence 4*, pages 209-254. American Elsevier, New York, 1969.

[Buchanan 78] B. G. Buchanan and T. Mitchell.
Model directed learning by production rules.
In D. A. Waterman and F. Hayes-Roth (editors), *Pattern-Directed Inference Systems*, pages 297-312. Academic Press, New York, 1978.

[Carbonell 78] J. G. Carbonell, Jr.
POLITICS: Automated ideological reasoning.
Cognitive Science 2(1):27-51, 1978.

[Carbonell 79] J. G. Carbonell.
The counterplanning process: a model of decision-making in adverse situations.
In *IJCAI-6*, pages 124-130. Tokyo, Japan, 1979.
Extended paper: Carnegie-Mellon Comp. Sci. Tech. Report.

[Carnegie-MellonUniversity 77]
Carnegie-Mellon University Speech Group.
Speech Understanding Systems: Summary of Results of the Five-Year Research Effort.
Technical Report, Carnegie-Mellon University Computer Science Department, Pittsburgh, PA, 1977.

[Darlington 76] J. Darlington and R. M. Burstall.
A system which automatically improves programs.
Acta Informatica 6:41-60, 1976.

[Davis 76] R. Davis and J. King.
An overview of production systems.
In E. W. Elcock and D. Michie (editors), *Machine Intelligence 8*, pages 300-332. Wiley, New York, 1976.

[Davis 77a] R. Davis, B. Buchanan, and E. H. Shortliffe.
Production rules as a representation for a knowledge based consultation system.
Artificial Intelligence 8:15-45, Spring, 1977.

[Davis 77b] R. Davis.
Interactive transfer of expertise: Acquisition of new inference rules.
In *IJCAI-5*, pages 321-328. Cambridge, MA, 1977.

[DeKleer 79] J. de Kleer.
The origin and resolution of ambiguities in causal arguments.
In *IJCAI-6*, pages 197-203. Tokyo, Japan, 1979.

[Dietterich 81] T. Dietterich and R. Michalski.
Inductive learning of structural descriptions.
Artificial Intelligence 16, 1981.

[Doyle 81] J. Doyle.
A truth maintenance system.
Artificial Intelligence 12(3), 1981.

[Duda 78] R. O. Duda, P. E. Hart, N. J. Nilsson, and G. Sutherland.
Semantic network representations in rule-based inference systems.
In D. A. Waterman and F. Hayes-Roth (editors), *Pattern-Directed Inference Systems*, pages 203-221. Academic Press, New York, 1978.

[Engelmore 79] R. Engelmore and A. Terry.
Structure and function of the CRYSALIS system.
In *IJCAI-6*, pages 250-256. Tokyo, Japan, 1979.

[Erman 75] L. D. Erman and V. R. Lesser.
A multi-level organization for problem solving using many, diverse, cooperating sources of knowledge.
In *IJCAI-4*, pages 483-490. Tbilisi, USSR, 1975.

[Fahlman 79] S. E. Fahlman.
NETL: A System for Representing and Using Real World Knowledge.
MIT Press, 1979.

[Feigenbaum 77] E. A. Feigenbaum.
The art of artificial intelligence: Themes and case studies of knowledge engineering.
In *IJCAI-5*, pages 1014-1029. Cambridge, MA, 1977.

[Fikes 68] R. E. Fikes.
A heuristic program for solving problems stated as nondeterministic procedures
PhD thesis, Carnegie-Mellon University, 1968.

[Fikes 71] R. E. Fikes and N. J. Nilsson.
STRIPS: a new approach to the application of theorem proving to problem solving.
Artificial Intelligence 2(3-4):189-208, 1971.

[Green 76] C. C. Green.
The design of the PSI program synthesis system.
In *Proceedings of the 2nd International Conference on Software Engineering*, pages 4-18. IEEE, San Francisco, 1976.

[Greenblatt 67] R. D. Greenblatt, D. E. Eastlake, and S. D. Crocker.
The Greenblatt chess program.
In *Fall Joint Computer Conference*, pages 801-810. AFIPS, 1967.

[Hayes-Roth 78a] F. Hayes-Roth, P. Klahr, J. Burge, and D. J. Mostow.
Machine methods for acquiring, learning, and applying knowledge.
Technical Report P-6241, The Rand Corporation, Santa Monica, CA, October, 1978.

[Hayes-Roth 78b] F. Hayes-Roth.
The role of partial and best matches in knowledge systems.
In D. A. Waterman and F. Hayes-Roth (editors), *Pattern-Directed Inference Systems*, pages 557-574. Academic Press, New York, 1978.

[Hayes-Roth 78c] F. Hayes-Roth and J. McDermott.
An interference matching technique for inducing abstractions.
Communications of the ACM 21(6):401-410, 1978.

[Hayes-Roth 80] F. Hayes-Roth, P. Klahr, and D. J. Mostow.
Knowledge acquisition, knowledge programming, and knowledge refinement.
Technical Report R-2540-NSF, The Rand Corporation, Santa Monica, CA, 1980.

[Hayes-Roth 81a] F. Hayes-Roth.
Proofs, refutations, and rectifications: heuristics for learning from empirical disconfirmations.
Technical Report N-1690-AF, The Rand Corporation, March, 1981.
Presented at the Carnegie-Mellon Workshop on Machine Learning, Pittsburgh, PA, July 1980.

[Hayes-Roth 81b] F. Hayes-Roth, P. Klahr, and D. J. Mostow.
Advice taking and knowledge refinement: an iterative view of skill acquisition.
In J. A. Anderson (editor), *Skill Acquisition and Development.* Erlbaum, 1981.
In press. Presented at 1980 Carnegie Symposium on Cognition, Pittsburgh, PA.

[Heidorn 74] G. E. Heidorn.
English as a very high level language for simulation programming.
In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, pages 91-100. Santa Monica, CA, 1974.

[Hendrix 75] G. G. Hendrix.
Expanding the utility of semantic networks through partitioning.
In *IJCAI-4*, pages 115-121. Tbilisi, USSR, 1975.

[Hewitt 75] C. Hewitt and B. Smith.
Toward a programmer's apprentice.
IEEE Transactions on Software Engineering 1:26-45, 1975.

[Hobbs 78] J. R. Hobbs and S. J. Rosenschein.
Making computational sense of Montague's intensional logic.
Artificial Intelligence 9:287-306, 1978.

[Kant 79] E. Kant.
A knowledge-based approach to using efficiency estimation in program synthesis.
In *IJCAI-6*, pages 457-462. Tokyo, Japan, 1979.

[Klahr 78] P. Klahr.
Partial proofs and partial answers.
Technical Report P-6239, The Rand Corporation, Santa Monica, CA, 1978.
To be presented at 4th Workshop on Automated Deduction, University of Texas, Austin, 1979.

[Korf 80] R. E. Korf.
Toward a model of representation changes.
Artificial Intelligence 14(1):41-78, 1980.

[Langley 79] P. Langley.
Rediscovering physics with BACON.3.
In *IJCAI-6*, pages 505-507. Tokyo, Japan, 1979.

[Lenat 78] D. Lenat and G. Harris.
Designing a rule system that searches for scientific discoveries.
In D. A. Waterman and F. Hayes-Roth (editors), *Pattern-Directed Inference Systems*, pages 25-51. Academic Press, New York, 1978.

[Lenat 79] D. B. Lenat, F. Hayes-Roth, and P. Klahr.
Cognitive economy in artificial intelligence systems.
In *IJCAI-6*, pages 531-536. Tokyo, Japan, 1979.

[Low 78] J. R. Low.
Automatic data structure selection: an example and overview.
Communications of the ACM 21(5):376-385, May, 1978.

[Luckham 79] D. Luckham and N. Suzuki.
Verification of Array, Record and Pointer Operations.
ACM TOPLAS 1(2):226-244, October, 1979.

[Manna 79] Z. Manna and R. Waldinger.
A deductive approach to program synthesis.
In *IJCAI-6*, pages 542-551. 1979.

[Marsh 70] D. Marsh.
Memo functions, the Graph Traverser, and a simple control situation.
In B. Meltzer and D. Michie (editors), *Machine Intelligence 5*, pages 281-300. American Elsevier, New York, 1970.

[Martin 77] N. Martin, P. Friedland, J. King, and M. Stefik.
Knowledge base management for experiment planning in molecular genetics.
In *IJCAI-5*, pages 882-887. Cambridge, MA, 1977.

[McCarthy 63] J. McCarthy et al.
LISP 1.5 Programmer's Manual.
MIT Press, Cambridge, MA, 1963.

[McCarthy 68] J. McCarthy.
The advice taker.
In M. Minsky (editor), *Semantic Information Processing*, pages 403-410. MIT Press, Cambridge, MA, 1968.

[McCarthy 77] J. McCarthy.
Epistemological problems of Artificial Intelligence.
In *IJCAI5*, pages 1038-1044. Cambridge, MA, 1977.

[Meehan 72] J. Meehan.
CANTUS.
1972.
Computer program to generate *cantus firmus*. Senior undergraduate honors project, Yale University.

[Meehan 79] J. R. Meehan.
An artificial intelligence approach to tonal music theory.
Technical Report 124A, Department of Information and Computer Science, University of California, Irvine, CA, June, 1979.

[Mitchell 77] T. M. Mitchell.
Version spaces: A candidate elimination approach to rule learning.
In *IJCAI-5*, pages 305-310. Cambridge, MA, 1977.

[Mitchell 81] T. M. Mitchell, P. E. Utgoff, and R. B. Banerji.
Learning Problem-Solving Heuristics by Experimentation.
In R. S. Michalski, J. G. Carbonell and T. M. Mitchell (editors), *Machine Learning.* Palo Alto, CA: Tioga Pub. Co., 1981.
To appear.

[Moore 71] J. Moore.
The design and evaluation of a knowledge net for Merlin.
PhD thesis, Carnegie-Mellon University, 1971.

[Moore 74] J. Moore and A. Newell.
How can Merlin understand?
In L. W. Gregg (editor), *Knowledge and Cognition*, pages 201-252. Lawrence Erlbaum Associates, New York, 1974.

[Moses 71] J. Moses.
Symbolic integration: the stormy decade.
Communications of the ACM 14(8):548-560, 1971.

[Mostow 78] D. J. Mostow.
Machine-aided heuristic programming: An AI approach to knowledge engineering.
October, 1978.
Unpublished dissertation proposal.

[Mostow 79a] D. J. Mostow and F. Hayes-Roth.
Operationalizing heuristics: some AI methods for assisting AI programming.
In *IJCAI-5*. Tokyo, Japan, 1979.
Condensed version of [Mostow 79b].

[Mostow 79b] D. J. Mostow and F. Hayes-Roth.
Machine-aided heuristic programming: A paradigm for knowledge engineering.
Technical Report Rand-N-1007-NSF, The Rand Corporation, Santa Monica, CA, 1979.

[Nelson 79] G. Nelson and D. C. Oppen.
Simplification by cooperating decision procedures.
ACM TOPLAS 1(2):245-257, October, 1979.

[Newell 57] A. Newell, J. Shaw, H. Simon.
Empirical explorations of the logic theory machine: A case study in heuristics.
In *Proceedings of the 1957 Western Joint Computer Conference*, pages 218-230. Western Joint Computer Conference, 1957.
Reprinted in E. Feigenbaum and J. Feldman (eds.) *Computers and Thought*, McGraw-Hill, 1963.

[Newell 60] A. Newell, J. Shaw, H. Simon.
Report on a general problem-solving program for a computer.
In *Proceedings of the International Conference on Information Processing*, pages 256-264. UNESCO, Paris, 1960.

[Newell 69] A. Newell.
Heuristic programming: Ill-structured problems.
In J. Aronofsky (editor), *Progress in Operations Research*, pages 363-414. Wiley, New York, 1969.

[Newell 72] A. Newell.
A theoretical exploration of mechanisms for coding the stimulus.
In A. W. Melton and E. Martin (editors), *Coding Processes in Human Memory*, pages 373-434. Winston, Washington, D. C., 1972.

[Nii 79] H. P. Nii and N. Aiello.
AGE (Attempt to Generalize): a knowledge-based program for building knowledge-based programs.
In *IJCAI-6*, pages 645-655. Tokyo, Japan, 1979.

[Nilsson 71] N. Nilsson.
Problem-Solving Methods in Artificial Intelligence.
McGraw-Hill, 1971.

[Novak 76] G. S. Novak.
Computer understanding of physics problems stated in natural language.
AJCL (microfiche):MF53, 1976.

[Rovner 76] P. D. Rovner.
Automatic representation selection for associative data structures.
PhD thesis, Harvard University, 1976.

[Sacerdoti 74] E. D. Sacerdoti.
Planning in a hierarchy of abstraction spaces.
Artificial Intelligence 5:115-135, 1974.

[Sacerdoti 77] E. D. Sacerdoti.
A Structure for Plans and Behavior.
Amsterdam: North-Holland, 1977.

[Salzer 69] F. Salzer and C. Schacter.
Counterpoint in Composition: the Study of Voice Leading.
McGraw-Hill, New York, 1969.

[Schank 77] R. C. Schank and R. P. Abelson.
Scripts, Goals, Plans, and Understanding.
Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1977.

[Simon 77] H. A. Simon.
Artificial intelligence systems that understand.
In *IJCAI-5*, pages 1059-1073. Cambridge, MA, 1977.

[Slagle 63] J. R. Slagle.
A heuristic program that solves symbolic integration problems in freshman calculus.
Journal of the ACM 10:507-520, 1963.

[Soloway 77] E. M. Soloway and E. M. Riseman.
Knowledge-directed learning.
SIGART Newsletter 63:49-55, 1977.
Proceedings of the Workshop on Pattern-Directed Inference Systems.

[Stefik 80] M. J. Stefik.
Planning with Constraints.
PhD thesis, Stanford University, 1980.

[Tappel 80] S. Tappel.
Some algorithm design methods.
In *Proceedings of the First Annual National Conference on Artificial Intelligence*, pages 64-67. American Association for Artificial Intelligence, Stanford, CA, 1980.

[Vere 78] S. A. Vere.
Inductive learning of relational productions.
In D. A. Waterman and F. Hayes-Roth (editors), *Pattern-Directed Inference Systems*, pages 281-295. Academic Press, New York, 1978.

[Waterman 70] D. A. Waterman.
Generalized learning techniques for automating the learning of heuristics.
Artificial Intelligence 1(1-2):121-170, Spring, 1970.

[Waterman 78] D. A. Waterman.
Exemplary programming in RITA.
In D. A. Waterman and F. Hayes-Roth (editors), *Pattern-Directed Inference Systems*, pages 261-279. Academic Press, New York, 1978.

[Wegbreit 75] B. Wegbreit.
Mechanical program analysis.
Communications of the ACM 18(9):528-539, September, 1975.

[Wilensky 78] R. Wilensky.
Understanding Goal-Based Stories.
PhD thesis, Yale University, Sept., 1978.

[Williams 72] T. G. Williams.
Some studies in game playing with a digital computer.
In H. A. Simon and L. Siklossy (editors), *Representation and Meaning: Experiments with Information Processing Systems*, pages 71-142. Prentice-Hall, Inc., 1972.

[Woods 75] W. A. Woods.
What's in a link: foundations for semantic networks.
In D. G. Bobrow and A. Collins (editors), *Representation and Understanding*, pages 35-82. Academic Press, 1975.

[Zadeh 79] L. A. Zadeh.
Approximate reasoning based on fuzzy logic.
In *IJCAI-6*, pages 1004-1010. Tokyo, Japan, 1979.

[Zobrist 73] A. L. Zobrist and F. R. Carlson, Jr.
An advice-taking chess computer.
Scientific American 228(6):95-105, June, 1973.

Index of Rules